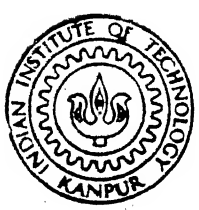
SOFTWARE IMPLEMENTATION OF DISCRETE CONVOLUTION AND FOURIER TRANSFORM ALGORITHMS

by

SATISH KAVETI



DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1989

SOFTWARE IMPLEMENTATION OF DISCRETE CONVOLUTION AND FOURIER TRANSFORM ALGORITHMS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

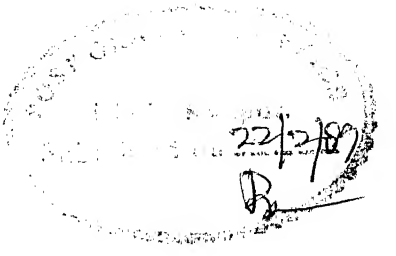
SATISH KAVETI

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1989



CERTIFICATE

This is to certify that this thesis titled " **SOFTWARE IMPLEMENTATION OF DISCRETE CONVOLUTION AND FOURIER TRANSFORM ALGORITHMS** " by Mr. Satish Kaveti has been carried out under my supervision and the same has not been submitted else where for a degree.

February, 1989.

(Prof. M.U.Siddiqi)

Department of Electrical Engineering
Indian Institute of Technology
Kanpur

ACKNOWLEDGEMENTS

It gives me immense pleasure in expressing my heartfelt thanks and deep sense of gratitute to my thesis supervisor, Prof. M. U. Siddiqi for his valuable guidance and constant encouragement.

I wish to thank my friends RamPrasad, Vemuri, Suneel, Banerjee, Sandhu, Verghese and Kushal for helping me at different stages of my thesis.

Satish Kaveti

ABSTRACT

In this thesis, we have given software implementation of various algorithms for discrete convolution and Fourier transform. Winograd short convolution based on the polynomial formulation and the matrix formulation have been implemented. Agarwal-Cooley algorithm is employed for convolving large sequences by breaking a long convolution into short convolutions. Short convolutions are computed by the matrix based formulation of Winograd algorithm. A better algorithm for multidimensional convolution is proposed.

The DFT algorithms implemented include the mixed-radix Cooley-Tukey algorithm, radix-2 fast Fourier transform and the Good-Thomas algorithm. Rader's algorithm which computes the Fourier transform of sequences of length equal to a power of an odd prime by converting it into convolutions is also implemented. The algorithm incorporates a scheme through which the DFT of a real sequence can be computed by real multiplications only, thereby achieving a reduction in computational complexity.

TABLE OF CONTENTS

CHAPTER ONE

I N T R O D U C T I O N

Computation of discrete Fourier transform has been gaining importance over the last few years. This is a direct consequence of the major role played by digital filtering and DFTs in digital signal processing and by the increasing use of digital processing techniques made possible by the declining cost of hardware. The motivation for developing fast algorithms is strongly rooted in the fact that the direct computation of length N convolutions and DFTs requires a number of operations proportional to N^2 which becomes rapidly excessive for long dimensions. This, in turn, implies an excessively large requirement for computer implementation of the methods.

1.1 Historical Background

Historically, the most important event in the fast algorithm has been the fast Fourier transform(FFT), introduced by Cooley and Tukey in 1965[1], which computes DFTs with a number of operations proportional to N·logN and therefore reduces drastically the computational complexity for large N. Algorithms were available earlier for fast computation of DFT for sequences of length equal to a power of 2, but the greatest emphasis, however, was on the computational economy that could be derived from the symmetries of the sine and cosine functions. The fast Fourier transform algorithm of Cooley and Tukey is more general in that it is applicable when N is a composite and not necessarily a power of 2.

Another important contribution to computing the DFT was

made by Thomas in 1963[2]. Like the transform algorithm of Cooley and Tukey it also achieved its economy by performing one - dimensional Fourier analysis by doing multidimensional Fourier analysis. However, the algorithms are different in the following ways :

1. In the Thomas algorithm the factors of N must be mutually prime.
2. In the Thomas algorithm the calculation is precisely multidimensional Fourier analysis with no intervening phase shifts or "twiddle factors" as they have been called.
3. The correspondences between the one - dimensional index and the multidimensional indexes are quite different.

It was shown by Good[3] that if N is composite, with mutually prime factors, i.e., $N = n_1 \cdot n_2 \cdot \ldots \cdot n_m$, one could do a one - dimensional Fourier analysis of N points by doing m - dimensional Fourier analysis on an m - dimensional, $r_1 \times r_2 \times \ldots \times r_m$ array of points. With these ideas put together and developed, Good's paper contains a full generalization of the Thomas prime factor algorithm.

The algorithm proposed by Cooley and Tukey was quite general, but later authors [4] have directed their attention to the special case of $N = 2^m$ than for the general case, and the restricted choice of N is adequate for the majority of applications. Gentleman and Sande [5] have extended the development of the general case and described the possible variations in organizing the algorithm. Further, Singleton [6]

presented an algorithm for computing the fast Fourier transform, based on the Cooley - Tukey algorithm for computing a transform step corresponding to an odd factor of N. The algorithm included an efficient method for permuting the results in place.

All the algorithms for the FFT require that the number of data samples, N, be highly composite to gain in terms of computation effort. Rader[7] showed that the DFT of N points, where N is a prime is essentially a cyclic convolution of length of (N-1). This allowed the DFT to be computed by means of a FFT algorithm, with the associated increase in speed, even though N is prime.

Convolution can be computed by DFT because of its cyclic convolution property. Therefore the FFT can be used to compute convolutions and thus the number of operations becomes proportional to $N \cdot \log N$. But the Fourier transform approach has an inherent disadvantage, due to the use of complex numbers even when the given sequences were real and the inability to give exact answers because of use of the irrational functions, namely sine and cosine.

For the computation of convolution the most significant contribution was that by Winograd in 1976. The algorithm proposed by Winograd[8,9] achieved a theoretical reduction of computational complexity over the FFT by a method which can be viewed as a converse of FFT, because it computes DFT as a convolution.

Another important factor in the development of new algorithms was the recognition that convolutions and DFTs can be viewed as operations over in finite rings and fields of integers and polynomials. This new point of view has allowed both

derivation of some lower computational bounds and design of new and improved computational techniques such as those based on polynomial transforms[10] and number theoretic transforms[11].

The Winograd algorithm while being efficient for small lengths of the data sequences becomes very cumbersome for large values of N. Agarwal & Cooley[12] gave an algorithm for converting a problem of convolving long sequences into convolution of shorter sub - sequences based on the Chinese Remainder Theorem.

1.2. Present work

The present work is concerned with the development and implementation of algorithms for discrete convolution and discrete Fourier transform. The treatment can be broadly divided into three sections :

1. Number theory and polynomial algebra.
2. Fast convolution algorithms.
3. Fast Fourier transform algorithms.

The first section is concerned with the development of algorithms for the mathematical tools which are used often in computation of discrete convolutions and discrete Fourier transforms.

Euclidean algorithm for finding the greatest common divisor and inverse of two integers have been implemented. Chinese Remainder theorem for integers is discussed along with its algorithm. An algorithm for finding the primitive root over GF(p) is proposed. Various operations on polynomials which are implemented include the basic operations like addition, multiplication, division and modulo on polynomials over real and

rational fields and integer ring. A better algorithm for division of polynomials over rational field which is less susceptible to overflow is proposed. The Euclidean algorithm is extended to polynomials over rational field and an algorithm for finding the inverse of a polynomial is implemented. The Chinese Remainder theorem for fast multiplication of polynomials when the modulo polynomial is monic is implemented. An algorithm for factorizing polynomials of the form (x^n-1) is also implemented.

The second section is concerned with algorithms for computation of discrete convolutions. Winograd algorithm for short convolutions is implemented. The polynomial and matrix based formulation of the Winograd algorithm have been dealt with. The matrix based formulation is used for development of algorithms for multidimensional convolution. A memory efficient and a faster algorithm for convolving large sequences using algorithms for short convolutions is implemented based on the Agarwal-Cooley algorithm. The convolution of two-dimensional sequences is done by nested convolution algorithm.

The third section deals with fast algorithms for discrete Fourier transform. The treatment starts with the mixed-radix Cooley Tukey algorithm and radix-2 Fast Fourier Transform along with it's algorithmic implementation. The Good Thomas algorithm for transforming a one-dimensional Fourier transform is implemented. The procedure includes an efficient algorithm for factorization of length N into relatively prime factors n_i, $i=1,2,\ldots,k$ where n_i can be expressed as $n_i = p_i^c$ and each p_i's are distinct. The Good Thomas algorithm utilizes the Cooley-Tukey FFT algorithm for n_i - point DFT. A computationally efficient

algorithm for DFT of real sequences with length equal to a power of an odd prime is implemented. The algorithm is based on Rader's algorithm for converting a DFT into a convolution and uses the Winograd algorithm for convolving the resulting sequences. The algorithm requires re-indexing if the input and output sequences and utilizes the symmetry property of the sine and cosine functions to minimize the computational effort.

The algorithms have been implemented on a IBM compatible personal computer. The language used for their implementation is Turbo Pascal Version 4.0.The appendix lists the various procedures available with brief comment on their usage.

1.3 Structure of the thesis

Chapter 2 gives a brief account of various mathematical concepts required for understanding the algorithms presented in this thesis. Algorithms for the frequently used ideas of number theory and polynomial algebra are also given.

Chapter 3 is devoted to the development of algorithms for discrete convolution.

Chapter 4 discusses various fast algorithms for the computation of discrete Fourier transform.

Finally, the last chapter concludes the thesis and discusses about the scope for future work.

Appendix A gives a list of all the Pascal procedures and functions implemented along with a brief comment on their usage.

CHAPTER TWO

A L G E B R A I C B A C K G R O U N D

This chapter introduces the necessary background required to understand various convolution and DFT algorithms in a simple, general way, with the intent of familiarization with the mathematical principles that are most frequently used in this thesis. After every section an algorithmic implementation of the methods discussed in that section is presented.

The chapter is broadly divided into two sections :

1. Elementary number theory
2. Polynomial algebra

In the elementary number theory the following topics have been discussed :

1. Euclidean algorithm
2. Chinese Remainder theorem for integers
3. Primitive roots of finite fields

In the polynomial algebra section the Euclidean algorithm and the Chinese Remainder Theorem are extended to polynomials. The relationship between polynomial algebra and convolution is introduced. A discussion on cyclotomic polynomials and the scheme for their generation follows.

2.1 Elementary Number Theory

2.1.1 Euclidean Algorithm

The greatest common divisor (gcd) can be found easily by a division algorithm known as the Euclidean algorithm[13].

Also, if a and b are relatively prime, i.e., if

$$GCD(a,b) = (a,b) = 1 \qquad (2.1)$$

then the Euclidean algorithm can be used to find an integer c such that

$$a \cdot c = 1 \bmod b \qquad (2.2)$$

Let us assume that a and b are positive integers. Dividing a by b yields

$$a = b \cdot q_1 + r_1 \ , \ r_1 < b \qquad (2.3)$$

by definition, $d=(a,b) \leq$ a or b. Therefore, if $r_1 = 0$, $b \mid a$ and $(a,b)=b$. If $r_1 \neq 0$, by continuation of this procedure, the the following system of equations are obtained :

$$\begin{aligned} b &= r_1 \cdot q_2 + r_2 \ , & r_2 &< r_1 \\ r_1 &= r_2 \cdot q_3 + r_3 \ , & r_3 &< r_2 \\ &\cdots\cdots\cdots\cdots\cdots & & \qquad (2.4) \\ r_{k-2} &= r_{k-1} \cdot q_k + r_k, & r_k &< r_{k-1} \\ r_{k-1} &= r_k \cdot q_{k+1} \end{aligned}$$

Since $r_1 > r_2 > r_3 > \ldots$. the last remainder is zero. Thus by last equation, $r_k \mid r_{k-1}$. The preceding equation implies that $r_k \mid r_{k-2}$, since $r_k \mid r_{k-1}$. Finally we obtain $r_k \mid b$ and $r_k \mid a$. Hence r_k is a divisor of a and b.

An important consequence of Euclidean Algorithm is that the GCD of two integers a and b is a linear combination of a and b. this can be seen by rewriting (2.3) and (2.4) as

$$\begin{aligned} r_1 &= a - b \cdot q_1 \\ r_2 &= b - r_1 \cdot q_2 \\ &\cdots\cdots\cdots\cdots & \qquad (2.5) \\ r_k &= r_{k-2} - r_{k-1} \cdot q_k \end{aligned}$$

The first equation shows that r_1 is a linear combination of a and

b. The second equation shows that r_2 is a linear combination of b and r_1 and therefore of both a and b. Finally the last equation implies that r_k is a linear combination of a and b. Since $r_k=(a,b)$, we have

$$(a,b) = ma + nb \qquad (2.6)$$

where m and n are integers. When a and b are relatively prime, (2.6) reduces to Bezout's relation

$$1 = ma + nb \qquad (2.7)$$

The value of m and n can be found from equation. (2.4) and (2.5).

From equation (2.7)

$$ma \equiv 1 \bmod b \qquad (2.8)$$

i.e., m is an inverse of a mod b.

The following set of recursive equations can be used to find m and n :

$$m_k = m_{k-2} - q_{k+1} \cdot m_{k-1}$$
$$n_k = n_{k-2} - q_{k+1} \cdot n_{k-1} \qquad (2.9)$$

where

$$r_k = r_{k-2} - r_{k-1} \cdot q_k$$
$$m_{-2} = 1 \ , \ n_{-2} = 0 \qquad (2.10)$$
$$m_{-1} = 0 \ , \ n_{-1} = 1$$
$$r_{-1} = a \ , \ r_0 = b.$$

The equations (2.9) are to be applied until $r_k = 1$.

2.1.1A Algorithm to find inverse : Given relatively prime integers a and b such that $a \geq b$, the following algorithm will find m,n satisfying (2.7)

1. Set $r_0 = b$ and $r_{-1} = a$. Set $m_{-2} = 1$ and $n_{-2} = 0$ and $m_{-1}=0$ and $n_{-1}=1$. Set k=1.

2. Divide r_{k-2} by r_{k-1} to obtain quotient q_k and remainder r_k. Put

$$m_{k-1} = m_{k-3} - q_k \cdot m_{k-2}$$

and $$n_{k-1} = n_{k-3} - q_k \cdot n_{k-2}$$

3. If $r_k = 1$ the algorithm terminates with $m = m_{k-1}$ and $n = n_{k-1}$. Otherwise set k=k+1 and return to step 2.

2.1.2 Chinese Remainder Theorem

In this section the problem of solving a set of simultaneous linear congruences with different moduli is discussed. The solution is given by the Chinese Remainder theorem(CRT). This theorem has extensive applications in various signal processing algorithms. The theorem is stated here without the proof.

Theorem 2.1 : Let m_i be k positive integers greater than 1 and relatively prime in pairs. The set of linear congruences

$$x \equiv r_i \bmod m_i \qquad (2.11)$$

has a unique solution modulo M, with $M = \pi\, m_i$.

Mathematically, given the residues r_i's, the solution of (2.11) can be found uniquely by

$$x = \sum_{i=1}^{k} (M / m_i) \cdot r_i \cdot T_i \qquad (2.12)$$

$$= \sum_{i=1}^{k} M_i \cdot N_i \cdot r_i$$

where $$M_i = M / m_i \qquad (2.13)$$

$$M_i \cdot N_i = 1 \bmod m_i$$

$$= 0 \bmod m_k, \quad k \neq i$$

The the integer N_i can be found by Euclid's algorithm, as N_i is the inverse of M_i.

2.1.2A Algorithm for Chinese Remainder Theorem : Given a set of relatively prime positive integers { m_i , i=1,...k } greater than 1, this algorithm will compute M_i and N_i satisfying (2.13).

1. Compute $M = \prod_{i=1}^{k} m_i$.
2. Set i = 1.
4. Set $M_i = M / m_i$. Use the Euclidean algorithm to find the inverse of M_i mod m_i. Let the inverse be called N_i.
4. If i ≤ k then return to step 3. Otherwise the algorithm terminates.

2.1.3 Primitive roots

Consider a prime field GF(p) where p is a prime number. In GF(p) one can always find an integer a < p such that $a^{(p-1)}=1$ and for any other integer j < (p-1) , $a^j \neq 1$. The integer a is said to be having an order (p-1) and is called the primitive element in GF(p). Some of the important theorems of number theory which will be used later are stated in the following paragraphs.

Definition : The number of integers that are smaller than m and relatively prime to m is denoted by Ø(m) and called Euler totient function.

When m is prime, with m=p, all integers smaller than p are relatively prime to p. Thus

$$Ø(p) = p-1$$

If $m = p^c$, the only numbers less than m and not prime with m are the multiples of p. Therefore

$$\emptyset(p^c) = p^{c-1}\cdot(p-1)$$

Theorem 2.2 If (a,m) = 1, then

$$a^{\emptyset(m)} = 1 \bmod m$$

Theorem 2.3 If (g,m) = 1 and $g^b \equiv 1 \bmod m$, the order of the integer g must divide b.

Theorem 2.4 If (g,m) = 1, the order r of the integer g must divide $\emptyset(m)$.

One of the important operation when working on finite fields is the search for a primitive element. For checking whether a the chosen integer n < p is primitive or not, one needs to check whether $n^q = 1$ or not, for every $q|(p-1)$.The following is the algorithm for search of the primitive element.

2.1.3A Algorithm for searching the primitive element : Given a prime number p, this algorithm will find the primitive root π in GF(p) such that

$$\pi^{p-1} = 1 \quad \text{and} \quad \pi^l \neq 1 \ , \ 0 < l < (p-1)$$

1. Factorize (p-1) and store the factors in f[i],i=1,...,m such that f[1]=2 and f[m]=(p-1); Set j =1.
2. Set j=j+1 and set k=j ; Set i=1.
3. If $k \neq 1$ then go to step 4. Otherwise, if i>m then the algorithm terminates with π=j else return to step 2.
4. Set $k=(j)^{f(i)}$ and set i=i+1. Return to step 3.

The step 4 of the last algorithm involves integer exponentiation. For large values of prime p, the algorithm takes sufficiently long time. The next algorithm will consider fast exponentiation.

2.1.3B Algorithm for fast exponentiation : Given positive integers a and b this algorithm will compute c such that

$$c = a^b$$

1. Set c=1
2. If b is odd then set c = c*a.
3. Set a=a² and set b=b div 2. If b=0 then the algorithm terminates , else return to step 2.

In the next section the Euclid's algorithm and the Chinese Remainder theorem shall be extended for polynomials. The algorithms for the various algebraic operations on polynomials are given.

Example 2.1: Consider p=17. To find out whether an integer k ∈ GF(p) is a primitive root or not it is necessary to find it's order. From Theorem 2.3 the order of k has to be a factor of (p-1)=16. The possible orders are 2,4,8 and 16.

Let k=2. It is not primitive because $2^8 = 256 \equiv 1 \bmod 17$ and thus, the order of 2 is 8.

Let k=3.

$$k^2 = 9 \bmod 17 = 9$$

$$k^4 = 81 \bmod 17 = 13$$

$$k^8 = 169 \bmod 17 = 16$$

$$k^{16} = 256 \bmod 17 = 1$$

Therefore, k is a primitive root in GF(17). The different powers of k can be computed using the fast exponentiation algorithm.

In the next section the Euclid's algorithm and the Chinese Remainder theorem are extended to polynomials. The algorithms for the various algebraic operations on polynomials are given.

2.2 Polynomial Algebra

Arithmetic on polynomials consists primarily of addition, subtraction, multiplication ; in some cases, further operations such as division, exponentiation, factoring, and taking the greatest common divisor are important. In this section a improved algorithm for division of polynomials over rational field is proposed. An algorithm for factorization of the polynomials of the form (x^n-1) is also suggested. The treatment is starts with the simpler algorithms .

The operation of addition and subtraction are straight forward. Multiplication of the polynomials is done by the rule,

$$(u_r \cdot x^r + \ldots + u_0) \cdot (v_s \cdot x^s + \ldots + v_0) = (w_{r+s} \cdot x^{r+s} + \ldots + w_0)$$

where

$$w_k = u_0 \cdot v_k + u_1 \cdot v_{k-1} + \ldots + u_{k-1} \cdot v_1 + u_k \cdot v_0$$

In the later formula u_i and v_j are treated as zero if $i>r$ or $j<s$. The division of polynomials is a relatively more difficult problem.

2.2.1 Division of polynomials

It is possible to divide one polynomial by another in essentially the same way that we divide one multi-precision integer by another, when arithmetic is done on polynomials over a field. The most important fields of coefficients that arise in applications are

1. The real and complex numbers
2. The rational numbers
3. Integers modulo p where p is a prime.

Given two polynomials u(x) and v(x) over a field, with

$v(x) \neq 0$, $u(x)$ can be divided by $v(x)$ to obtain a quotient polynomial $q(x)$ and a remainder polynomial $r(x)$ satisfying the conditions

$$u(x) = q(x) \cdot v(x) + r(x) \qquad (2.14)$$

The following algorithm can be used to divide two polynomials over a field. The algorithm considers the most straight forward method.

2.2.1A Algorithm for dividing polynomials over a field : Given polynomials $u(x)$ and $v(x)$ of degree m and n respectively, this algorithm finds the polynomial $q(x)$ and $r(x)$ satisfying (2.14)

1. Do step 2 for $k=m-n, m-n-1, \ldots, 0$; then the algorithm terminates with $(r_{n-1}, \ldots, r_0) = (u_{n-1}, \ldots, u_0)$.
2. Set $q_k = (u_{n+k} / v_n)$, and then set $u_j = u_j - q_k \cdot v_{j-k}$ for $j= n+k-1, n+k-2, \ldots, k$ (The latter operation amounts to replacing $u(x)$ by $u(x) - q_k \cdot x^k \cdot v(k)$, a polynomial of degree $< n+k$).

While the above algorithm works perfectly well for division of real polynomials , for polynomials over integer rings suitable modifications are required because of non - existence of an inverse. Of course, situations when an integer polynomial $u(x)$ is perfectly divisible by integer polynomial $v(x)$, and ,when $v(x)$ is a monic polynomial the above algorithm continues to hold valid.

An algorithm was proposed by Knuth[13] to overcome the problem of division of integer polynomials. It can be seen that the above algorithm requires explicit division only by the leading coefficient of $v(x)$, written as $l(v)$, and that the step 2 is carried out exactly $(m-n+1)$ times; thus if $u(x)$ and $v(x)$ start with integer coefficients, and if the algebraic structure over

which the operations are done are rational numbers, then the only denominators that appear in the coefficients of q(x) and r(x) are divisors of $l(v)^{m-n+1}$. This suggests that the polynomials q'(x) and r'(x) can be found such that

$$l(v)^{m-n+1} \cdot u(x) = q'(x) \cdot v(x) + r'(x) \quad (2.15)$$

where m=deg(u) and n=deg(v), for any polynomials u(x) and $v(x) \neq 0$, provided that $m \geq n$. The values q(x) and r(x) can be found from q'(x) and r'(x) as follows,

$$q(x) = q'(x) \,/\, l(v)^{m-n+1}$$

$$r(x) = r'(x) \,/\, l(v)^{m-n+1} \quad (2.16)$$

The integer divisors in the above expressions are stored as separate variable as the field of operation is rational. The above algorithm was adequate for cases when l(v) and (m-n) are small integers, for other cases the algorithm results in an early overflow.

An algorithm is proposed by which such a problem could be avoided. For convenience of explanation, the content and primitive part of a polynomial are defined. The content of a polynomial u(x), abbreviated as cont(u), is the gcd of the coefficients of u(x),and the primitive part of u(x), abbreviated as pp(u), is u(x)/cont(u). The following steps outline the algorithm.

2.2.1B Division of integer polynomials over rational field : Given polynomials u(x) and v(x) over integer ring, this algorithm will find q(x) and r(x) over rational field satisfying (2.14).

1. Set num=cont(u) and den=cont(v) ; set v(x)=pp(v) and set u(x)=pp(u),set t=1.
2. Do step 3 for k = m-n,m-n-1,...,0 (The integer indicates the degree of the quotient polynomial); then

go to step 4.

3. Set x = gcd(l(u) l(v)) ; Do the following settings :

$s = v_n / n$, $t = t \cdot s$, $q_k = u_{n+k} / x$,

$u_j = u_j \cdot v_n/x - u_{n+k} \cdot v_{j-k}/x$ for j=n+k-1, ... 1,0

$q_i = q_i \cdot v_n/x$ for i=m-n,....,k+1 .

4. Set u(x)= (num/t)·u(x) and q(x)=(num/(t·den))·q(x),the algorithm terminates with r(x)=u(x).

In the above algorithm the division in step 3 is done over integer ring, but in step 4 the common multipliers and divisors are stored in different variables instead of actual multiplication and division.

Example 2.2 : Consider polynomials u(x) and v(x) such that

$$u(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$

$$= (1\ 0\ 1\ 0\ -3\ -3\ 8\ 2\ -5)$$

$$v(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

$$= (3\ 0\ 5\ 0\ -4\ -9\ 21)$$

When using (2.15) and (2.16), u(x) is replaced by $u(x) \cdot l(v)^{m-n+1} = u(x) \cdot 3^3 = 27 \cdot u(x)$. So the new value of u(x) is

$$u(x) = (27\ 0\ 27\ 0\ -81\ -81\ 216\ 54\ -135)$$

Now u(x) can be divided by v(x) to obtain

$$q'(x) = (9\ 0\ -6)$$

$$r'(x) = (-15\ 0\ 5\ 0\ -15)$$

It is seen that the quotient and the remainder have a common factor 3. Also the starting value of u(x) has large coefficients even though the given polynomial has small coefficients.

When using Algorithm 2.2.1B for dividing, the resulting coefficients are of smaller magnitude, but the algorithm requires

computation of gcd at every step.

The values of u(x),q(x) and r(x) after each iteration are indicated in the table below :

$$v(x) = (3\ 0\ 5\ 0\ -4\ -9\ 21)$$

$$u(x) = (1\ 0\ 1\ 0\ -3\ -3\ 8\ 2\ -5)$$

u(x)	r(x)	q(x)
3 0 3 0 -9 -9 24 6 -15	0 -2 0 -5 0 3 6 -15	1
0 -2 0 -5 0 3 6 -15	-2 0 -5 0 3 6 -15	1 0
-6 0 -15 0 9 18 -45	0 -5 0 1 0 -3	3 0 -2

From , algorithm 2.2.1B

$$r(x) = (-5\ 0\ 1\ 0\ -3)/9$$

and $$q(x) = (3\ 0\ -2)/9$$

2.2.2 Euclidean Algorithm for polynomials : In this section the set of equations for the Euclid's algorithm will be presented without dwelling into the the theory.

Consider polynomials M(x) and m(x) with integer coefficients such that they are relatively prime to each other ; if, the deg(M(x)) ≤ deg(m(x)), then the Euclid's algorithm can be used to find N(x) such that,

$$M(x)\cdot N(x) = 1 \bmod m(x) \qquad (2.17)$$

The following set of recursive equations can be used to find N(x) :

$$N_k(x) = N_{k-2}(x) - q_{k+1}(x)\cdot N_{k-1}(x) \qquad (2.18)$$

where

$$r_k(x) = r_{k-2}(x) - r_{k-1}(x)\cdot q_k(x)$$

$$N_{-2}(x) = 1\ ,\quad N_{-1}(x) = 0\ , \qquad (2.19)$$

$$r_{-1}(x) = a \ , \ r_0(x) = b$$

The equations (2.9) are to be applied until $r_k(x) = 1$.

2.2.3 Chinese Remainder Theorem over polynomial fields : The convolution algorithm can be alternatively stated as a product of polynomials. Therefore, if $\{s_i\}$ is obtained from a convolution of $\{g_i\}$ and $\{d_i\}$ then the operation can be written as

$$s(x) = g(x) \cdot d(x) \bmod m(x) \qquad (2.20)$$

Suppose that m(x) is a product of k polynomials $m_i(x)$ having no common factors, i.e.,

$$m(x) = \prod_{i=1}^{k} m_i(x) \qquad (2.21)$$

Since each of these polynomials $m_i(x)$ is relatively prime with all the other polynomials $m_u(x)$, it has an inverse modulo every other polynomial. This means that we can extend the Chinese Remainder Theorem to the ring of polynomials modulo m(x) and therefore express uniquely s(x) as a function of the polynomials $s_i(x)$ obtained by reducing s(x) modulo the various polynomials $m_i(x)$. The Chinese Remainder theorem is then expressed as

$$s(x) = \sum_{i=1}^{k} s_i(x) \cdot M_i(x) \cdot N_i(x) \bmod m(x) \qquad (2.22)$$

where, for every value of u and i

$$M_i(x) = m(x)/m_i(x) \qquad (2.23)$$

and

$$M_u(x) \cdot N_u(x) \equiv 0 \bmod m_i(x), \ i \neq u$$

$$\equiv 1 \bmod m_u(x)$$

The validity of the above can be easily verified. To compute $N_u(x)$

the Euclidean algorithm can be used.

The residues $s_i(x)$ can be evaluated using the residues $g_i(x)$ and $d_i(x)$ by

$$s_i(x) = g_i(x) \cdot d_i(x) \bmod m_i(x) \qquad (2.24)$$

2.2.3A **Algorithm for CRT over polynomial rings :** Given real polynomials, d(x) and g(x) and an integer polynomial m(x), this algorithm will find s(x) satisfying (2.20) using (2.21) - (2.24).

1. Factorize m(x), store the factors as $m_i(x)$, i=1,2,...,k
2. For i=1 to k , compute $M_i(x)=m(x)/m_i(x)$.
3. For i=1 to k , use the Euclidean algorithm to compute $N_i(x)$ such that

   $$M_i(x) \cdot N_i(x) = 1 \bmod m_i(x).$$

4. Compute the residues $d_i(x)$ and $g_i(x)$. Evaluate $s_i(x)$ satisfying (2.24).
5. Set s(x) = 0.
6. Do step 7 for i=1 to k , then the algorithm terminates.
7. Set $s(x) = s(x) + M_i(x) \cdot N_i(x) \cdot s_i(x)$.

Example 2.3 : Consider

$$m(x) = x^4 - 1$$
$$= (x-1) \cdot (x+1) \cdot (x^2-1)$$
$$= m_1(x) \cdot m_2(x) \cdot m_3(x)$$

From (2.23)

$$M_1(x) = (x^3 + x^2 + x + 1)$$
$$M_2(x) = (x^3 - x^2 + x - 1)$$
$$M_3(x) = (x^2 - 1)$$

Using Euclidean algorithm the inverse $N_i(x)$, i=1,2,3 satisfying (2.23) can be computed. From (2.17),(2.18) and (2.19),

$$N_1(x) = 1/4$$

$$N_2(x) = -1/4$$

$$N_3(x) = -1/2 \ .$$

The operation which is done very often is finding the residues. In the next paragraph an algorithm is given for finding the residue when the modulo polynomial is monic. This does not impose serious restriction, as for most of the signal processing applications it is monic.

2.2.3B Algorithm for evaluation of residue polynomial when the modulo polynomial is monic : Given a real polynomial s(x) and an integer monic polynomial m(x), this algorithm will find s'(x) such that

$$s'(x) = s(x) \bmod m(x)$$

1. If deg(s(x)) < deg(m(x)), the algorithm terminates.
2. Do step 3 for k = n-1,n-2,...,m ; the algorithm terminates with s'(x)=s(x).
3. Set $s_j = s_j - s_k \cdot c_{j-k+m}$ for j=k-1,k-2,...,k-m.

2.2.4 Factorization of polynomials : Factorization of polynomials is not a simple problem. Efficient algorithms exist [13] for factorization of polynomials over finite field due to Berlekamp[16].

In this section the attention is restricted to the class of polynomials of the form x^n-1. For most of the situations this is sufficient because polynomials of different form are used only for linear convolution. Under such circumstances , m(x) is decided on the basis of the factors , so that the residues $s_i(x)$ can be found with least effort. This implies that for those cases when linear convolution is desired, the factors of m(x) are pre - determined.

2.2.5 Cyclotomic polynomials : The polynomial x^n-1 can be written in terms of its prime factors

$$x^n-1 = p_1(x)\cdot p_2(x)\cdot p_3(x)\cdot \ldots . p_k(x) \quad (2.25)$$

If the field over which factorization is done is the field of rationals, the prime factors are easy to find. These factors are called cyclotomic polynomials.

For small n, The cyclotomic polynomials are easy to find by factoring x^n-1. Clearly

$$Q_1(x) = x-1$$

Few more polynomials shall be factored to see the general pattern. Let n=2. Then

$$x^2-1 = (x-1)(x+1) = Q_1(x)\cdot Q_2(x)$$

Let n=3.Then

$$x^3-1 = (x-1)(x^2+x+1) = Q_1(x)\cdot Q_2(x)$$

Let n=4.Then

$$x^4-1 = (x-1)(x+1)(x^2+1) = Q_1(x)\cdot Q_2(x)\cdot Q_4(x)$$

At each step, only one new polynomial occurs. The general case is described by the following theorems. The proofs of the theorems shall not be discussed. An interested reader may refer to Blahut[14].

Theorem 2.5 For each n

$$\deg(Q_n(x)) = \emptyset(n) \quad (2.26)$$

where $\emptyset(n)$ is the totient function.

Theorem 2.6 For each n

$$x^n-1 = \prod_{k|n} Q_k(n) \quad (2.28)$$

In the next paragraph an algorithm for the factorization of polynomials of the form x^n-1 is suggested. For signal processing

applications this operation can be done off-line and the factors can be stored at definite memory locations.

2.2.5A Algorithm for factorization of polynomials of form x^n-1: Given a positive integer $n \geq 1$, this algorithm will find the factors of x^n-1 over rational field.

1. Let k be an array of integer and Factor an array of polynomials over integer ring ; Set k[1]=1 and set Factor[1]=(x-1). Set l=2 and j=1.
2. If $l \neq n$ then set l=l+1; Set $t(x)=x^l-1$.
3. for i=1 to j do

   { If k[i]|l then do

   { Set t(x) = t(x) / Factor[i] }

   }
4. Set j=j+1; Set k[j]=l and set Factor[j]=t(x)
5. Set l=l+1 ; if $l \leq n$ return to step 2 otherwise the algorithm terminates with Factor containing the factors of x^n-1.

In this chapter the basic mathematical tools needed for description of the various convolution and DFT algorithms were discussed. In the next chapter various convolution algorithms will be discussed.

CHAPTER THREE

D I S C R E T E C O N V O L U T I O N

3.1 Introduction

The calculation of finite digital convolution

$$y_i = \sum_{k=0}^{N-1} h_{i-k} \cdot x_k \qquad (3.1)$$

has extensive applications in both the general - purpose computers and specially constructed digital processing devices. It is used to compute the auto - and cross - correlation functions, to design and implement finite impulse response and infinite impulse response digital filters, to solve difference equations, and to compute power spectra.

While the calculation of the convolution according to the defining formula (3.1) would require a number of multiplications and additions proportional to N^2 for large N, use of Fast Fourier Transform (FFT) has been able to reduce this to $O(N \cdot \log N)$ operations when N is a power of 2. To be more specific, consider the problem where h_i, i= .... , -1,0,1,... is a periodic sequence of period N so that $h_i = h_{N+i}$. Then the discrete Fourier transform(DFT)

$$X_n = \sum_{k=0}^{N-1} x_k \cdot e^{(-2 \cdot \pi \cdot j \cdot n \cdot k/N)} \qquad (3.2)$$

has the property that the DFT's H_n, X_n and Y_n, n=0,1,2,...,N-1, of the three sequences h_k, x_k and y_k, k=0,1,2,...,N-1, respectively,

are related by

$$Y_n = X_n \cdot H_n \qquad n=0,1,2,\ldots,N-1 \qquad (3.3)$$

Since the FFT algorithm enables one to calculate the DFT in $O(N \cdot \log(N))$ operations, the entire convolution operation requires $O(N \cdot \log(N))$ operations.

But still there remains few problems to be resolved before using the FFT algorithm for the computation of convolutions. One of the most important is when the h_i's and x_i's are from the integer ring. The computationally efficient DFT method involves intermediate quantities, i.e., sines and cosines, which are irrational numbers, thereby making exact results impossible on a digital machine. Also, there is a need for doing complex arithmetic, even if the convolutions are real.

To resolve the first problem Agarwal & Burrus[11] suggested number theoretic transforms to implement fast digital convolution. The transforms are defined on finite fields and rings of integers with arithmetic carried out modulo an integer.

One more approach which does not require working in complex fields relies strongly on representation of convolution as polynomial multiplication and Chinese Remainder theorem. This approach was introduced in Chapter 2. Even though, the algorithm does not require operations on complex numbers, the computation of convolution requires a number of subsidiary operations like factorization of modulo polynomial, computation of $M_i(x)$ and $N_i(x)$, evaluation of the residues,etc., which reduces the efficiency of the algorithm. The Winograd algorithm for convolving data sequences is based on the Chinese Remainder Theorem, and expresses the convolution operation as a sequence of pre -

additions, multiplications and post additions.

3.2 Winograd Algorithm

As already stated, the Winograd algorithm is based on the Chinese remainder theorem. Therefore, for sake of convenience in understanding the algorithm the Chinese Remainder theorem is restated :

If

$$s(x) = g(x) \cdot d(x) \bmod m(x) \qquad (3.1)$$

and the modulo polynomial m(x) is factorizable as

$$m(x) = \prod_{i=1}^{k} m_i(x) \qquad (3.2)$$

then the problem of convolution can be broken as a sequence of smaller convolutions

$$s_i(x) = d_i(x) \cdot g_i(x) \bmod m_i(x) \qquad i=1,2, \ldots ,k \qquad (3.3)$$

where

$$d_i(x) = d(x) \bmod m_i(x)$$
$$g_i(x) = g(x) \bmod m_i(x) \qquad (3.4)$$

The polynomial s(x) can be obtained from its residues $s_i(x), i=1,2, \ldots ,k$ by the Chinese Remainder Theorem, given as

$$s(x) = \sum_{i=1}^{k} s_i(x) \cdot M_i(x) \cdot N_i(x) \qquad (3.5)$$

where

$$M_i(x) = m(x)/m_i(x) \quad \text{and}$$
$$M_i(x) \cdot N_i(x) = 1 \bmod m_i(x) \qquad (3.6)$$

To see that the above set of equations actually do reduce

the computational complexity , an example is considered.

Let the number of data points, N=8, then

$$m(x) = x^8 - 1$$
$$= (x-1)\cdot(x+1)\cdot(x^2+1)\cdot(x^4+1)$$
$$= Q_1(x)\cdot Q_2(x)\cdot Q_4(x)\cdot Q_8(x)$$
$$= m_1(x)\cdot m_2(x)\cdot m_3(x)\cdot m_4(x)$$

Direct convolution would require $N^2=64$ multiplications. When using the Chinese Remainder Theorem the number multiplications would be 1+1+4+16=22. Thus, by breaking the problem of convolution of large sequences into convolution of smaller sequences a reduction in the computational effort can be achieved.

The coefficients of the product polynomial $d_i(x)\cdot g_i(x)$ gives the values of the non-cyclic n_j-point convolution of the coefficients of $d_i(x)$ and $g_i(x)$ [$n_j=\deg(m_j(x))+1$]. Then according to discussion before, $s_i(x)$ is the result of reducing this polynomial mod $m_i(x)$. It can be easily seen that $d_i(x)\cdot g_i(x)$ can be computed by multiplying linear combinations of the coefficients of $d_i(x)$ by the linear combinations of the coefficients of $g_i(x)$. These coefficients are, in turn, linear combinations of d_i's and g_i's, respectively. The set of products, so formed is, therefore, of the form

$$S = (Bg)\cdot(Ad) \qquad (3.7)$$

where $\cdot$ denotes element by element multiplication.

Substituting the $s_i(x)$ in the CRT results in formulas for the s_i's as linear combinations of the above mentioned. Thus, one obtains the form

$$s = CS \qquad (3.8)$$

Equations (3.7) and (3.8) give alternative scheme to convolve two

sequences.

To reduce the order of the A and B matrices, the Cook-Toom algorithm [14], other systematic procedures and even manual manipulation can be used.

Example 3.1 : Consider 4 - point cyclic convolution of two real sequences $\{ g_i, i=0,1,\ldots,4 \}$ and $\{d_i, i=0,1,\ldots,5\}$. In terms of polynomials whose coefficients are the sequences involved, this corresponds to

$$s(x) = g(x) \cdot d(x) \bmod (x^4-1)$$

As stated earlier in Example 2.3

$$M_1(x) = (x+1)(x^2+1) \; ; \; N_1(x) = 1/4$$

$$M_2(x) = (x-1)(x^2+1) \; ; \; N_2(x) = -1/4$$

$$M_3(x) = (x^2-1) \; ; \; N_3(x) = -1/2$$

The reduced polynomials

$$d_i(x) = d(x) \bmod m_i(x)$$

are

$$d_1(x) = d_0^{\,1} = d_0 + d_1 + d_2 + d_3$$

$$d_2(x) = d_0^{\,2} = d_0 - d_1 + d_2 - d_3$$

$$d_3(x) = d_0^{\,3} + d_1^{\,3}x = (d_0 - d_2) + (d_1 - d_3)x$$

The equations for

$$g_i(x) = g(x) \bmod m_i(x)$$

are exactly the same form as those for $d_i(x)$. The relation

$$s_i(x) = d_i(x) \cdot g_i(x) \bmod m_i(x)$$

is, in terms of the coefficients of $d_i(x)$ and $g_i(x)$,

$$s_0^{\,1} = d_0^{\,1} \cdot g_0^{\,1}$$

$$s_0^{\,2} = d_0^{\,2} \cdot g_0^{\,2}$$

$$s_0^{\,3} = d_0^{\,3} \cdot g_0^{\,3} - d_1^{\,3} \cdot g_1^{\,3}$$

$$s_1^3 = d_0^3 \cdot g_1^3 + d_1^3 \cdot g_0^3$$

The calculation of $s_3(x)$ is exactly like complex multiplication and carried out as though $x = \sqrt{-1}$. The Cook - Toom algorithm can be used to compute s_0^3 and s_1^3, in 3 instead of 4 multiplications. The result is that it is necessary to compute the five products

$$m_0 = d_0^1 \cdot g_0^1$$

$$m_1 = d_0^2 \cdot g_0^2$$

$$m_2 = d_0^3 \cdot (g_0^3 + g_1^3)$$

$$m_3 = g_0^3 \cdot (d_0^3 - d_1^3)$$

$$m_4 = g_1^3 \cdot (d_0^3 + d_1^3)$$

In terms of these, the s_i^j's are

$$s_0^1 = m_0$$

$$s_0^2 = m_1$$

$$s_0^3 = m_2 - m_4$$

$$s_1^3 = m_2 - m_3$$

The polynomial $s_i(x)$ whose coefficients are given above are then substituted in the Chinese Remainder theorem

$$s(x) = \sum_{i=1}^{3} s_i(x) \cdot M_i(x) \cdot N_i(x)$$

to give the final result,

$$s_0 = (m_0 + m_1)/4 + (m_2 - m_4)/2$$

$$s_1 = (m_0 - m_1)/4 + (m_2 - m_3)/2$$

$$s_2 = (m_0 + m_1)/4 - (m_2 - m_4)/2$$

$$s_3 = (m_0 - m_1)/4 - (m_2 - m_3)/2$$

If g_i's are the filter coefficients then they can be assumed to be fixed and used repeatedly for many d_i sequences.

Accordingly, the computation can be simplified by redefining the m_k's and combining the 1/2 and 1/4 factors with the g_i's. In terms of matrices the algorithm can written as,

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 2 & 0 & -2 & 0 \\ 2 & -2 & -2 & 2 \\ 2 & 2 & -2 & -2 \end{bmatrix} \cdot (1/4)$$

Let

$$G = Bg$$

and $$D = Ad$$

Compute S such that

$$S = G \cdot D$$

where · indicates component wise multiplication.

Also,

$$C = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & -1 & 1 & -1 & 0 \\ 1 & 1 & -1 & 0 & 1 \\ 1 & -1 & -1 & 1 & 0 \end{bmatrix}$$

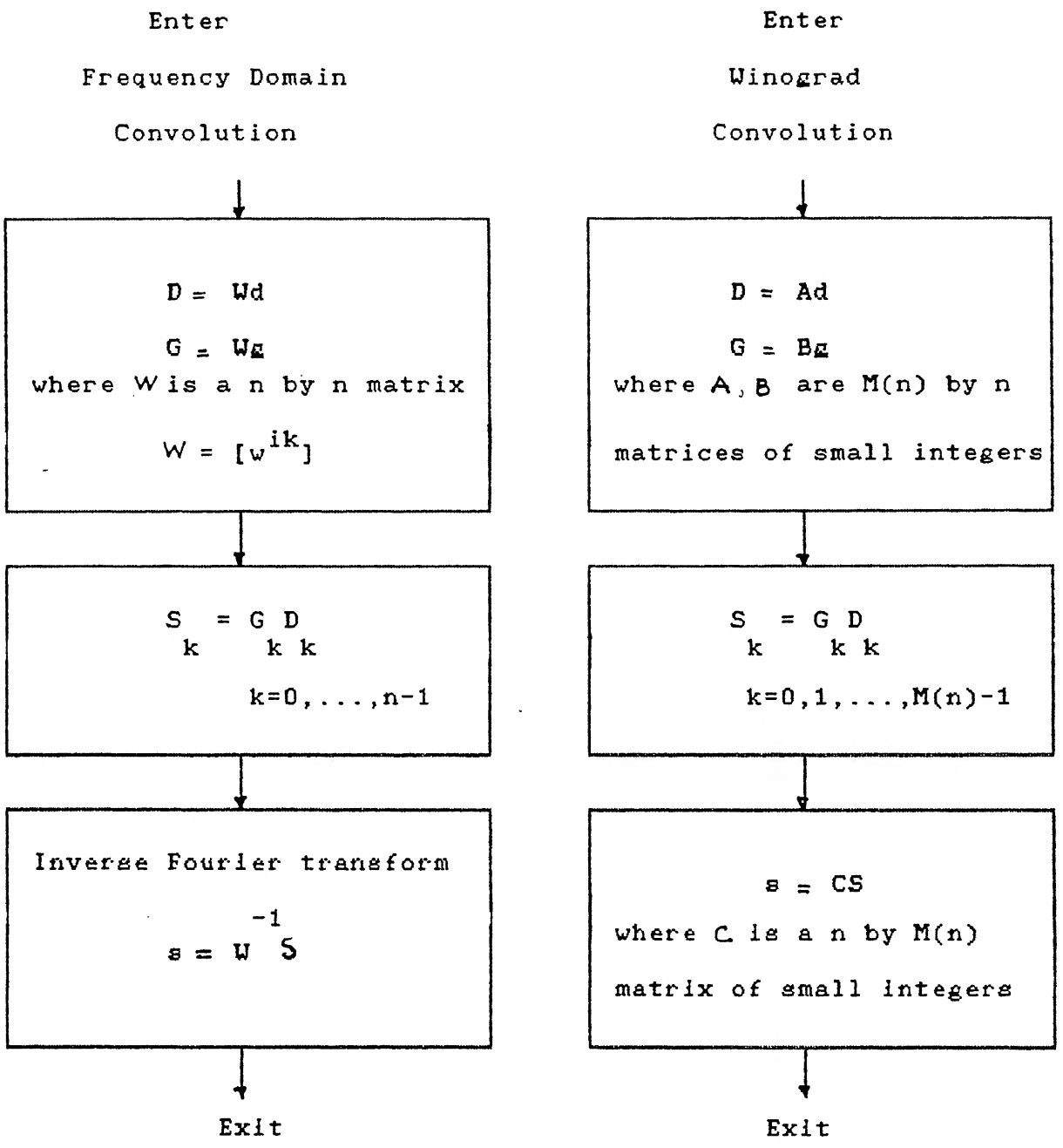


FIG 1. COMPARISON OF TWO CONVOLUTION METHODS

The result can be obtained by

$$s = CS$$

One of the important implication of representing convolution in terms of matrices,is its similarity to the DFT approach . Figure 1 gives an instructive comparison of a Winograd convolution algorithm and a convolution computed by using a discrete Fourier transform. The transform,in this case, is unlike the DFT, is not a square transform but instead a rectangular transformation. The transform notation of the Winograd algorithm has important consequences in multi - dimensional convolution algorithms.

3.3 Multidimensional_Convolutions

3.3.1 Introduction

This section is intended to familiarize the reader with the multidimensional convolution. The treatment is elementary and for a detailed study the reader is advised to refer to Blahut[20].

The convolution of two multidimensional sequences $g(n_1,\ldots,n_r)$, $d(n_1,\ldots,n_r)$ over a field is described by

$$s(k_1,k_2,\ldots,k_r) = \Sigma\, g(n_1,n_2,\ldots,n_r)\cdot d(k_1-n_1,\ldots,k_r-n_r) \qquad (3.9)$$

The field F can be field of real numbers R, the field of complex numbers C, of any other finite or infinite field.

An equivalent definition may be given in terms of the polynomial form of the above sequences

$$s(x_1,\ldots,x_r) = d(x_1,\ldots,x_r)\cdot g(x_1,\ldots,x_r) \qquad (3.10)$$

For cyclic convolution (3.10) becomes

$$s(x_1,\ldots,x_r) = d(x_1,\ldots,x_r)\cdot g(x_1,\ldots,x_r) \quad (3.11)$$
$$\text{mod } (x_1^{N1}-1),,\ldots,(x_r^{Nr}-1)$$

The next section describes an algorithm for multidimensional convolution.

3.3.2 Nested Convolution Algorithm : For ease of analysis, only two - dimensional case will be considered. For convolutions of higher dimensions the following treatment can be easily extended.

The two - dimensional cyclic convolution can be also written in terms of polynomials. Thus

$$s(x,y) = g(x,y)\cdot d(x,y) \ (\text{mod } x^{n'}-1)(\text{mod } x^{n''}-1) \quad (3.12)$$

where n' and n" need not be equal. A two-dimensional cyclic convolution satisfies the cyclic convolution theorem. If D and G are the two dimensional Fourier transforms of d and g, respectively, and $S_{k',k''} = G_{k',k''}\cdot D_{k',k''}$, then S is the two-dimensional Fourier transform of s. Hence a two-dimensional cyclic convolution can be computed by using two-dimensional FFT algorithm. One can also use direct methods.

The two-dimensional convolution may be easier to understand when it is written as a convolution of polynomials. It can easily be seen that a two-dimensional convolution can be written as

$$s_{i'}(y) = \Sigma\ g_{(i'-k')}(y)\cdot d_{k'}(y) \quad (3.13)$$

The convolution algorithms discussed in the previous sections, though derived in an arbitrary field, actually are valid in some rings, such as polynomial rings. Hence the convolution of polynomials can be computed by any of these fast algorithms. Addition in the fast algorithm become addition of polynomials, and

multiplications become multiplications of polynomials. Those multiplications of polynomials are themselves convolution and in turn can be computed by any fast algorithm. Thus one way to compute a two-dimensional convolution is obtained by nesting a fast algorithm for one-dimensional convolution inside another fast algorithm for one - dimensional convolution. The next paragraph presents an algorithm for fast two-dimensional convolution based on the nesting algorithm.

3.3.2A Algorithm for two-dimensional convolution : Given two N' and N" array

$$d = \{ d_{i',i"},\ i'=0,1,\ldots,N'-1\ ;i"=0,1,\ldots,N"-1 \}$$

$$g = \{ g_{i',i"},\ i'=0,1,\ldots,N'-1\ ;i"=0,1,\ldots,N"-1 \}$$

the algorithm computes a new array

$$s = \{ s_{i',i"},\ i'=0,1,\ldots,N'-1\ ;i"=0,1,\ldots,N"-1 \}$$

satisfying

$$s_{i',i"} = \sum_{k'=0}^{N'-1} \sum_{k"=0}^{N"-1} g_{(i'-k'),(i"-k")} \cdot d_{k',k"}$$

$$i' = 0,1,2,\ldots,N'-1$$

$$i" = 0,1,2,\ldots,N"-1$$

1. Multiply every row of d by matrix A" and every row of g by the matrix B".
2. Multiply every column of the transformed d array by A' and every column of the transformed g array by B'. Let the resulting arrays be D and G respectively.
3. Multiply D and G component wise. Let the resulting array be called S.
4. Multiply every column of S by C' and then multiply

every row by C". The resulting array will contain the elements of s.

In the above algorithm, the transformation was first done along the rows and then along the columns. Alternatively it can be done along the columns first and then along the rows. The number of multiplications would be the same for both the implementations but the number of additions would differ. To minimize the number of additions, it has been shown [13], the transformation should be done first along the dimension for which $M(n_i)-n_i/A(n_i)$ is minimum, where $M(n_i)$ and $A(n_i)$ is the number of multiplications and additions respectively for a convolution of length n_i .

3.3.3 Agarwal Cooley Convolution Algorithm

For large convolutions, the algorithms derived from the interpolation methods become complicated and /efficient. The Agarwal-Cooley algorithm greatly simplifies the computation of long cyclic convolution by using a one-dimensional to multidimensional mapping suggested by Good[3,15], combined with the nesting approach proposed by Agarwal and Cooley[12].

In this section after an introductory treatment of the Agarwal Cooley algorithm, a fast algorithm is implemented for one-dimensional convolution of length N where $N = n_1 \cdot n_2 \cdot \ \ldots \ n_m$ and n_i's are relatively prime. The transformation is done in place without actually transforming the data into multi - dimensional form.

Consider again, the problem of computing the cyclic convolution

$$s_i = \sum_{k=0}^{N-1} d_{(i'-k')} \cdot g_k \qquad (3.14)$$

where N is a composite number

$$N = n_1 \cdot n_2 \qquad (3.15)$$

with mutually prime factors n_1 and n_2. This permits to define the one - to - one mapping

$$i \equiv (i_1, i_2) \qquad (3.16)$$

where i_1 and i_2 are defined by the congruence relations

$$\begin{aligned} i_1 &= i \bmod n_1 , & 0 \leq i_1 \leq n_1 \\ i_2 &= i \bmod n_2 & 0 \leq i_2 \leq n_2 \end{aligned} \qquad (3.17)$$

The CRT gives a unique solution i to the congruences (3.17) which is given by

$$i = i_1 \cdot M_1 \cdot N_1 + i_2 \cdot M_2 \cdot N_2 \quad 0 \leq i < N \qquad (3.18)$$

where

$$\begin{aligned} M_1 &= N / n_1 \\ \text{and} \quad M_2 &= N / n_2 \end{aligned} \qquad (3.19)$$

and

$$\begin{aligned} M_1 \cdot N_1 &= 1 \bmod n_1 \\ M_2 \cdot N_2 &= 1 \bmod n_2 \end{aligned} \qquad (3.20)$$

Substituting (3.18) for i and a similar expression for k in terms of (k_1, k_2), the convolution (3.14) can be written

$$s_{i1,i2} = \sum_{k_2=0}^{n_2-1} \sum_{k_1=0}^{n_1-1} d_{(i1-k1),(i2-k2)} \cdot g_{k1,k2} \qquad (3.21)$$

where the indices of $d_{i1,i2}$ are understood to be taken mod n_1 and n_2 respectively. Once the data has been converted into a two - dimensional form the algorithm for fast two - dimensional convolution discussed earlier can be used. The result is converted

back to one - dimension by using (3.18). To minimize the number of multiplications the following should be true :

1. Multiplication with the pre - addition matrix A and B should be done first along the dimension for which [M(n) - n]/A(n) is the least, where M(n) is the number of multiplications for n - point convolution and A(n) is the number of additions ; n is the number of data points along that dimension.
2. The C operators should be placed in the reverse order to that used for A and B operator.

The above treatment can be easily generalized for the multidimensional case.

3.3.4 General Multifactor Algorithm for the Agarwal Cooley Algorithm

In this section an algorithm is suggested for convolution of sequences of length N where N can be factored into r relatively prime factors. The algorithm is based on the Agarwal - Cooley algorithm for transforming the one - dimensional data into multidimensional form and the Winograd's algorithm for computing smaller convolutions. One of the important features of the algorithm is that, in contrast with the algorithms available in the literature, it does not require generation of the A,B and C matrices, thereby, in effect, reduces the memory requirement of the algorithm.

3.3.4A Algorithm for large convolution : Let d,g be vectors of size N, this algorithm computes the vector s such that

$$s_i = \sum_{k=0}^{N-1} d_{i-k} \cdot g_k \qquad (3.22)$$

1. Factorize N into relatively prime factors n_i's such that

$$N = n_1 \cdot n_2 \cdot \ldots \cdot n_r \qquad (3.23)$$

2. Choose the mixed radix number system $(n_r,\ldots,n_2,n_1)$. The number in this radix number system is given by

$$i' = \sum_{k=1}^{r} \text{weight}[k] \cdot i_k \qquad (3.24)$$

where

$$\text{weight}[1] = 1 \quad \& \quad \text{weight}[k] = \text{weight}[k-1] \cdot n_{k-1} \quad k=2,\ldots,r \qquad (3.25)$$

3. For i=0 to N-1 find the residues i_k, k=1,2,...,r such that

$$i_k = i \bmod m_k$$

Use the equation (3.24) to compute i'. Scramble the elements of the input vector such that the ith element becomes i'th element.

4. Set highmult=N , highorder=1, loworder=1 and i=0.

5. Set i=i+1;If i>r then go to step 10 else continue.

6. Set loworder=loworder/n_i .

7. Repeat step 8 for highbit=0 to highorder-1, later go to step 5 .

8. Repeat step 9 for lowbit=0 to loworder-1, later jump to step 7 .

9. for element=0 to n_i-1 compute

pos=highmult*highbit+loworder*element+lowbit.

Set buff1[element]=d[pos] and set buff2[element]=g[pos].

Transform the elements of the buff1 and buff2 vector through Winograd transformation. The new size of the Buff1 and Buff2 vector is $M(n_i)x1$.

10. Set highmult=highmult/n_i*$M(n_i)$.

11. Put the transformed elements back in a new vector using the radix system as $(M(n_1),..,M(n_i),n_{i+1},..,n_r)$.

for element=0 to n_i-1 compute

pos=highmult*highbit+loworder*element+lowbit.

Set d'[pos]=buff1[element] and set g'[pos]=buff2[element].

12. Set d=d' and g=g'.

13. Set highorder=highorder*$M(n_i)$.

After the operations the new radix system would be $(M(n_1),M(n_2),...,M(n_r))$. The result vector would be of size $M(n_1)\cdot M(n_2)\cdot ... \cdot M(n_r)$. Multiply the components of the resulting vectors component wise.

14. Carry out the inverse transformation similar to the forward transformation.

15. Permute the elements of the result to obtain the elements.

In the above algorithm the steps which require further analysis is the step 1, when the N is factored, and step 15 when the results are permuted back in natural order.

3.3.4B Algorithm for factorization into primes : Given a positive integer n, this algorithm finds the prime factors $p_1,p_2,...,p_t$ of n and also finds integers $n_1,n_2,...,n_r$ such that n_i's are relatively prime such that

$$n = p_1 \cdot p_2 \cdot p_3 \cdot \ldots p_t \qquad (3.26)$$

also $n = n_1 \cdot n_2 \cdot ... \cdot n_m$ where n_i's are relatively prime.

Each of the factor n_i can be expressed as

$$n_i = p'^{c_i}_i \qquad (3.27)$$

where each of the primes p'_i are distinct. The method makes use of an auxiliary sequence of "trial divisors"

$$2 = d_0 < d_1 < d_2 < \ldots$$

which includes all prime numbers $\leq \sqrt{n}$.

1. Set t=0 , k=0, and let l be the number of prime numbers smaller than $\sqrt{n}$. Set m=1.Set n_m=1.
2. If n=1, the algorithm terminates.
3. Set r = n mod d_k
4. If $r \neq 0$, go to step 7.
5. Increase t by 1, set $p_t=d_k$ and set n=n div d_k. Set $n_m=n_m \cdot d_k$.
6. If r = n mod d_k =0 then return to step 5.Otherwise set m=m+1 and n_m=1.
7. If k < l, then increase k by 1 and return to step 3.
8. Set t by 1,set p_t=n, and set n_m=n ; terminate the algorithm.

3.3.4C **Algorithm for permuting the results in natural order** : The major steps in the algorithm are

1. Converting the number to the mixed-radix representation.
2. Use of the Chinese Remainder Theorem to obtain the solution of linear congruences.

1. Set i=0.
2. Set t=N ; for i=1 to m do the following steps :

$N = n_1 \times n_2 = 2 \times 3 = 6$

Scrambling

Compute $i_1 = i \bmod n_1$

$i_2 = i \bmod n_2$

Find $i' = 3i_1 + i_2$

Permute the input vectors g and d such that

$x_{i'} = x_i$

Pre addition

Do the operation of pre addition on the scrambled d and g vectors along i_1. Then do pre addition along i_2.

The resulting vectors are of size $M(n_1) \cdot M(n_2) = 8$. Let the resulting vector be called G and D respectively.

Multiplication

Multiply D ang G component wise and the result be called S.

Post addition

Do post additions along i_2 and then along i_1 on the S vector. The resulting vector, s, will be of length N=6.

Unscrambling

Permute the elements such that

$x_i = x_{i'}$

FIG 2. MULTIFACTOR CONVOLUTION

1. Set k=1 and set l=m.

2. Set $t=t/n_k$.

3. Set weight[k]=t

4. Set k=k+1 and set l=l-1; if l ≥ 1 return to step 2; otherwise go out of the loop.

3. Use the Chinese Remainder theorem discussed in Chapter 2 to find the coefficients M_i and N_i such that

$$i = \sum_{k=0}^{m} M_k \cdot N_k \cdot i_k$$

Let $S_k = M_k \cdot N_k$

4. Do step 5 for i = 0 to N-1 then the algorithm terminates with the result in the buffer array.

5. Set j'=0 ; for k=m downto 1 do the following steps.

   a. Set t=i mod n_k. Set j=j div n_k .

   b. Set j' = j' + S_k.

   Put the element at the ith position at the j'th position in a buffer array.

Example 3.2 : Consider cyclic convolution of sequences of length N=6. It can be factored as

$$N = m_1 \times m_2 = 2 \times 3$$

$$M_1 = 3 \ , \ M_2 = 2$$

$$N_1 = 1 \ , \ N_2 = 2$$

The different operations are summarized in Figure 2.

CHAPTER FOUR

D I S C R E T E F O U R I E R T R A N S F O R M

4.1 Discrete Fourier Transform

The DFT is defined by

$$X_r = \sum_{k=0}^{N-1} x_k \exp(-2\pi jrk/N) \qquad (4.1)$$

where X_r is the r^{th} co-efficient of the DFT and x_k denotes the k^{th} sample of the time series which consists of N samples. The x_k's can be complex numbers and X_r's are almost always complex. For notational convenience (4.1) is often written as

$$X_r = \sum_{k=0}^{N-1} x_k W_N^{rk} \qquad (4.2)$$

where

$$W_N = \exp(-2\pi j/N)$$

There exists the usual inverse of the DFT and, because the form is very similar to that of the DFT, the same algorithm can be used to compute it. The inverse of (4.2) is

$$x_l = (1/N) \sum_{r=0}^{N-1} X_r W_N^{-rl} \qquad (4.3)$$

From (4.2), we can see that calculating all the N-DFT coefficients would require N^2 multiplications, which is unacceptable for large problems ; fast algorithms are needed.

4.2 Cooley Tukey Algorithm

This section describes an algorithm for computing the Fast Fourier transform based on a method proposed by Cooley and Tukey. As in their algorithm, the dimension n of the transform is factored (if possible) and n/p elementary transforms of dimension

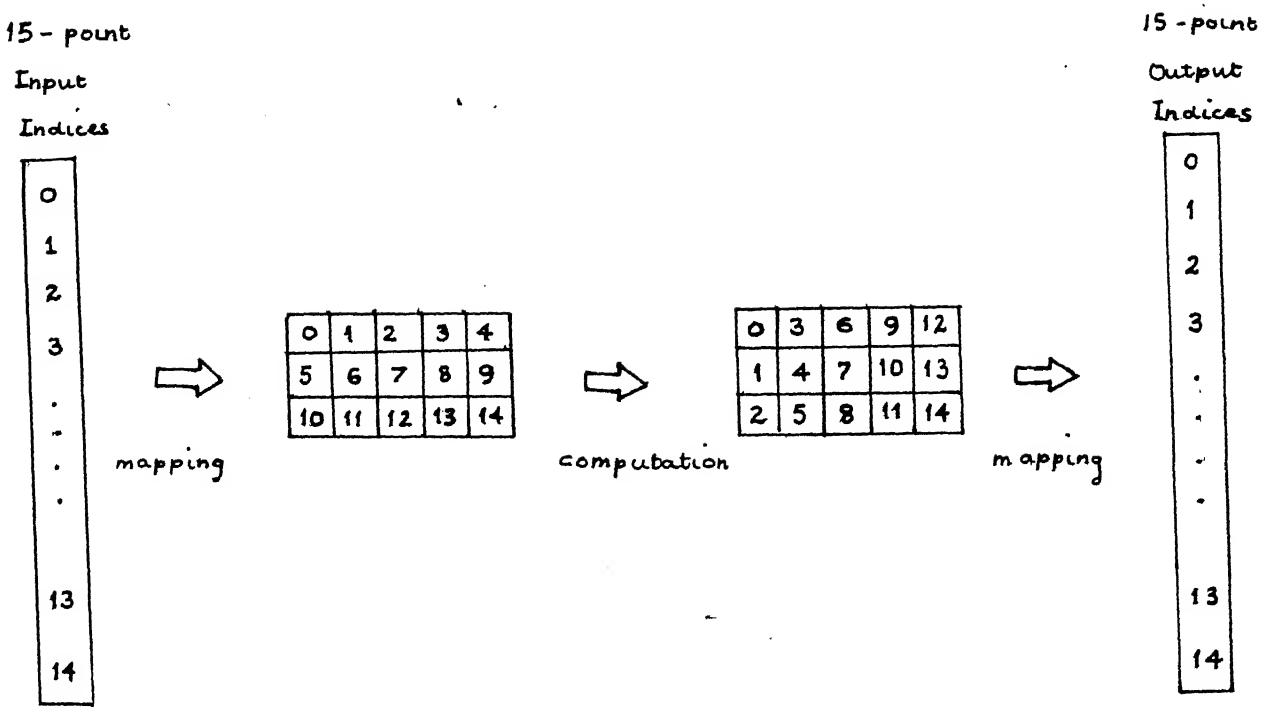


FIG 3 EXAMPLE OF COOLEY - TUKEY SHUFFLING

p are done for each factor p of n. The fast Fourier transform, when computed in place, requires a final permutation step to arrange the results in normal order.

4.2.1 Theory

From (4.2), we have

$$X_r = \sum_{k=0}^{N-1} x_k W_N^{rk} \qquad (4.4)$$

Let $N=n_1 \cdot n_2$ and let the indices r and k be expressed as

$$r = n_1 \cdot r_2 + r_1$$
$$\text{and} \qquad k = n_2 \cdot k_1 + k_2 \qquad (4.5)$$

Then we can write (4.4) as

$$X_{r2,r1} = \sum_{k_2=0}^{n_2-1} \sum_{k_1=0}^{n_1-1} x_{k1,k2} \cdot W_N^{(n1 \cdot r2 + r1) \cdot (n2 \cdot k1 + k2)}$$

$$= \sum_{k_2=0}^{n_2-1} W_{n2}^{r2k2} \cdot W_N^{r1k2} \sum_{k_1=0}^{n_1-1} x_{k1,k2} \cdot W_{n1}^{r1k1} \qquad (4.6)$$

From the above equation, it is easy to see that Cooley-Tukey FFT can be visualized as a mapping of one - dimensional array into two - dimensional array as shown in Figure 3 for N=15. The computation consists of an n_1 - point discrete Fourier transform on each column, followed by an element - by - element complex multiplication throughout the new array by W_N^{r1k2}, followed by n_2-point discrete Fourier transform on each row.

Equations (4.5) and (4.6) can be generalized for the multidimensional case. The Fourier transform along the most significant digit in (4.5) should be done first and transform along the least significant at the end. This is evident from

(4.5), in which k_1 is the most significant digit and in (4.6) Fourier transform along k_1 is done first. From equations (4.5) and (4.6) it is clear that if the Fourier transform is computed in place, i.e. the transformed samples are put back at the same position as the input elements themselves, then the result would be in bit reversed order. This is because the mixed - radix number system for the input and output index are reverse of each other. Therefore to obtain the output sequence in normal order, permutation operation is required.

Algebraically, if $N = n_1 \cdot n_2 \cdot n_3 \cdot \ldots \cdot n_m$ (4.7)

then

$$k = n_2 \cdot n_3 \cdot \ldots n_m \cdot k_1 + \ldots . + n_m \cdot k_{m-1} + k_m \quad (4.8)$$

$$r = n_1 \cdot n_2 \cdot \ldots \cdot n_{m-1} \cdot r_m + \ldots . + n_1 \cdot r_2 + r_1 \quad (4.9)$$

While taking the Fourier transform take the n_1 - point Fourier transform first and the n_m - point Fourier transform last. When doing the permutation operation put the element at location j, where

$$j = n_2 \cdot n_3 \cdot \ldots n_m \cdot j_1 + \ldots \ldots + n_m \cdot j_{m-1} + j_m \quad (4.10)$$

at location j', where

$$j' = n_1 \cdot n_2 \cdot \ldots \cdot n_{m-1} \cdot j_m + \ldots . + n_1 \cdot j_2 + j_1 \quad (4.11)$$

Example 4.1 Let us consider a sequence of length N = 15.

It can be factored as $N = n_1 \times n_2 = 3 \times 5$

The mixed radix representation of the input index i is be given as

$$i = (i_1 , i_2) = 5i_1 + i_2 , \quad i_1 = 0,1,2$$

$$i_2 = 0,1,2,3,4$$

To compute the discrete Fourier transform it is neccessary to first do a 3 - point DFT. The index of different elements to obtained by changing i_1 while keeping i_2 fixed. The Fourier transform of the three elements chosen is taken and the result is put back at the same position. The sampe operation is done for all values of i_2 from 0 to 4.

In the second stage of operation requires computation of 5-point Fourier transform. As the Fourier transform is done along i_2, which is the least significant digit, the samples whose DFT is desired are at consecutive positions.

For unscrambling the resulting sequence the ith element should be shifted to i'th position, where

$$i = 5i_1 + i_2 , \quad i_1 = 0,1,2$$

$$i_2 = 0,1,2,3,4$$

and $$i' = 3i_2 + i_1 .$$

4.3 Radix - 2 FFT

This section is concerned with the development of a variation of the algorithm that is better adapted to parallel processing in a special purpose machine. The work is based on the algorithm proposed by Pease[8].

For the purpose of parallel processing in a special purpose machine, we require that the operations be organized in a few levels, each of which involves a set of elementary operations that can be done simultaneously. Preferably each level should involve only a single type of elementary process, so that the entire level can be initiated by a single command. Also there should be as few as possible distinct types of elementary operations, so that the

parallel capability required shall be as simple as possible. For example, in the original algorithm each "pass" involves a different grouping of the data into pairs. If possible, it is preferable to use only a single pattern of pairings which could then be "wired in".

4.3.1 Fast Fourier Transform

In the matrix notation, the Fourier transform can be expressed as

$$X = T \cdot x \tag{4.12}$$

where

$$x = [\, x_0 \quad x_1 \quad x_2 \quad .... \quad x_{N-1} \,]^T$$

$$X = [\, X_0 \quad X_1 \quad X_2 \quad .... \quad X_{N-1} \,]^T$$

where x_k and X_r are related by (4.1).

The matrix T is called the finite Fourier transform. Writing w as the principal Nth root of unity, the coefficients of T is given by

$$(T)^{rs} = w^{rs} \tag{4.13}$$

To simplify notation, only the exponent of w will be preserved. That is, we write k in place of w^k. Then T can be written as

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 & ..... & 0 \\ 0 & 1 & 2 & 3 & & N-1 \\ 0 & 2 & 4 & 6 & & 2(N-1) \\ 0 & 3 & 6 & 9 & & 3(N-1) \\ . & . & . & . & . & . \\ 0 & (N-1) & 2(N-1) & ..... & & (N-1)^2 \end{bmatrix} \tag{4.14}$$

In this notation, multiplication of entries becomes addition. The expression for T can be further reduced by noting that each term may be replaced by its principal value (between 0 to N-1), modulo N.

The transformation used by Cooley and Tukey accomplishes the same thing as T, except that the output is obtained in permuted order - specifically in digit reversed order, where the rows are numbered from 0 to N-1, as expressed in the base of the prime factor being used (in the present case, in binary notation). The transformation of this form is given, in general, by $T'=Q\cdot T$, where Q is the appropriate permutation matrix.

The matrix T' can be obtained directly from T by permuting the rows of T as specified. For example, for N=8, after reducing all the terms modulo 8, we find that T is reduced exponent notation is given by

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & 3 & 6 & 1 & 4 & 7 & 2 & 5 \\ 0 & 4 & 0 & 4 & 0 & 4 & 0 & 4 \\ 0 & 5 & 2 & 7 & 4 & 1 & 6 & 3 \\ 0 & 6 & 4 & 2 & 0 & 6 & 4 & 2 \\ 0 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix} \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix} \qquad (4.15)$$

where the binary row designation appears on the right. Then the modified transform is given by T' as follows.

$$T' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 4 & 0 & 4 & 0 & 4 \\ 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & 6 & 4 & 2 & 0 & 6 & 4 & 2 \\ 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\ 0 & 5 & 2 & 7 & 4 & 1 & 6 & 3 \\ 0 & 3 & 6 & 1 & 4 & 7 & 2 & 5 \\ 0 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix} \quad \begin{matrix} 000 \\ 100 \\ 010 \\ 110 \\ 001 \\ 101 \\ 011 \\ 111 \end{matrix} \qquad (4.16)$$

The key to the fast Fourier transform lies in the fact that T_N' (when $N = 2^n$) can be first partition and then factored :

$$T' = \begin{bmatrix} T'_{N/2} & T'_{N/2} \\ (T'_{N/2})\,K & -(T'_{N/2})\,K \end{bmatrix} \qquad (4.17)$$

$$= \begin{bmatrix} T'_{N/2} & \emptyset \\ \emptyset & T'_{N/2} \end{bmatrix} \cdot \begin{bmatrix} I & I \\ K & -K \end{bmatrix}$$

$$= \begin{bmatrix} T'_{N/2} & \emptyset \\ \emptyset & T'_{N/2} \end{bmatrix} \cdot \begin{bmatrix} I & \emptyset \\ \emptyset & K \end{bmatrix} \cdot \begin{bmatrix} I & I \\ I & -I \end{bmatrix}$$

where, in exponent notation, K = diag(0 1 2 3 ... N-1) and $\emptyset$ indicates the null matrix of appropriate dimension.

The minus sign that occurs in (4.17) applies to the values of the sub - matrices that follow it, and not to the exponent coefficient that may actually be used. Since w is the nth root of unity, $w^{N/2}$ is a square root of unity or the value (-1). Hence the minus sign can be translated to exponent notation by adding N/2 to

the exponent coefficients involved.

It should also be noted that T_N' is written in terms of w, the principal Nth root of unity. $T_{N/2}'$ would normally be written in terms of principal (N/2)nd root, which is w^2. Hence, if equation (4.17) is actually written out, the entries in $T_{N/2}'$ will be twice those that would occur were it written in terms of the (N/2)nd root.

The algorithm can now be iterated by applying the factorization of Equation (4.17) to each occurrence of $T_{N/2}'$, and so on. If the process is carried to completion the Fast Fourier Transform is obtained.

For the purpose of parallelization, the process given suffers from the defect that a coefficient in a given location is combined with the coefficients in different location each time the sum and difference are formed. For example, if equation(4.17) is carried over to next stage of iteration,

$$T' = \begin{bmatrix} T'_{N/4} & \emptyset & \emptyset & \emptyset \\ \emptyset & T'_{N/4} & \emptyset & \emptyset \\ \emptyset & \emptyset & T'_{N/4} & \emptyset \\ \emptyset & \emptyset & \emptyset & T'_{N/4} \end{bmatrix} \cdot$$

$$\begin{bmatrix} I & \emptyset & \emptyset & \emptyset \\ \emptyset & K' & \emptyset & \emptyset \\ \emptyset & \emptyset & I & \emptyset \\ \emptyset & \emptyset & \emptyset & K' \end{bmatrix} \cdot \begin{bmatrix} I & I & \emptyset & \emptyset \\ I & -I & \emptyset & \emptyset \\ \emptyset & \emptyset & I & I \\ \emptyset & \emptyset & I & -I \end{bmatrix} \cdot$$

$$\begin{bmatrix} I & \emptyset \\ \emptyset & K \end{bmatrix} \cdot \begin{bmatrix} I & I \\ I & -I \end{bmatrix}$$

The last factor, which is the first operator applied to the data, forms the sum and differences of coefficients in the first half with the corresponding terms in the second. The third factor forms the sum and differences of the coefficients in the first quarter with corresponding terms in the second, and the coefficients in the third quarter, with those in the fourth. This prevents "wired-in" data paths, or else it requires an excessive number of such paths.

Therefore, there is a need to modify the algorithm to avoid this difficulty.

4.3.2 Modified Fast Fourier Transform

The recursion formula of equation (4.17) can be written in the form

$$T'_N = (T'_{N/2} \times I_2) D_N (I_{N/2} \times T_2') \qquad (4.18)$$

where D_N is the NxN diagonal matrix, quasidiag(I K), and T'_2 is given by

$$T'_2 = \begin{bmatrix} 0 & 0 \\ 0 & N/2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (4.19)$$

where in the later matrix the coefficients are the actual values, not exponents. The multiplication sign in (4.18) indicates the direct, or Kronekar product of the sub - matrices, indexed first on the indices of the first component and then on the indices of the second. For example, if $A = (a_{ij})$, $B = (b_{kh})$, with all the indices being either 0 or 1, then

$$A \times B = \begin{bmatrix} a\ b & a\ b & a\ b & a\ b \\ 00\ 00 & 01\ 00 & 00\ 01 & 01\ 01 \\ a\ b & a\ b & a\ b & a\ b \\ 10\ 00 & 11\ 00 & 10\ 01 & 11\ 01 \\ a\ b & a\ b & a\ b & a\ b \\ 00\ 10 & 01\ 10 & 00\ 11 & 01\ 11 \\ a\ b & a\ b & a\ b & a\ b \\ 10\ 10 & 11\ 10 & 10\ 11 & 11\ 11 \end{bmatrix}$$

The Kronekar product can be combined with matrix multiplication through the formula,

$$(A \times B)(C \times D) = (AC) \times (BD)$$

provided that the dimensions of the matrices are compatible. In particular, it can written that

$$(AB \times I) = (AB) \times I^2 = (A \times I)(B \times I)$$

or, more generally, if A,B,C,... are all kxk matrices, then

$$(ABC \ldots\ldots) \times I = (A \times I)(B \times I)(C \times I) \ldots$$

Using this formula, we can apply eqn(4.18) iteratively and obtain, when $N=2^n$,

$$\begin{aligned} T'_N = {} & (\, T'_2 \times I_2 \times \ldots. \times I_2 \,)\cdot Q_1 \cdot \\ & (\, I_2 \times T'_2 \times \ldots. \times I_2 \,)\cdot Q_2 \cdot \\ & \ldots\ldots\ldots\ldots \\ & (\, I_2 \times I_2 \times \ldots. \times T'_2 \,) \end{aligned} \qquad (4.20)$$

where each of the cross - products includes n factors, of which (n-1) are 2x2 identity and one is T'_2, and the Q_i are the diagonal

matrices. For example, for N=8, $T'_8 = (T'_4 \times I_2)D_8(I_4 \times I_2)$, and since $I_4 = I_2 \times I_2$ and

$$T'_4 = (T'_2 \times I_2)D_4(I_2 \times T'_2)$$

we find that

$$\begin{aligned} T'_8 &= ([(T'_2 \times I_2)D_4(I_2 \times T'_2)] \times I_2)D_8(I_4 \times I_2) \\ &= ((T'_2 \times I_2 \times I_2)(D_4 \times I_2)(I_2 \times I_2 \times T'_2))D_8(I_2 \times I_2 \times T'_2) \\ &= ((T'_2 \times I_2 \times I_2)Q_1(I_2 \times I_2 \times T'_2))Q_2(I_2 \times I_2 \times T'_2) \end{aligned} \qquad (4.21)$$

where $Q_1 = D_4 \times I_2$, $Q_2 = D_8$.

For the purpose of parallelization the difficulty now with eqn(4.20) is that T'_2 occurs in different locations in the various factors. There exists a permutation operator, P, such that

$$P(T'_2 \times A_{N/2})P^{-1} = A_{N/2} \times T'_2 \qquad (4.22)$$

If so, we can write each of the product terms in (4.20) in terms of a single one of them, say the first

$$\begin{aligned} C &= T'_2 \times I_2 \times I_2 \times I_2 \times \ldots \times I_2 \\ &= T'_2 \times I_{N/2} \end{aligned} \qquad (4.23)$$

By considering the behavior of the product operators on an arbitrary vector, a possible P is the "ideal schuffle", in which

$$\begin{aligned} P \cdot [x_0 \; x_1 \; \ldots \; x_{N/2-1} \; x_{N/2} \; \ldots \; x_{N-1} \;]^T \\ = [x_0 \; x_{N/2} \; x_1 \; x_{N/2+1} \; \ldots \; x_{N/2-1} \; x_{N-1} \;]^T \end{aligned} \qquad (4.24)$$

or, for N=8, for example

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (4.25)$$

where the entries, in this case, are the actual values, not exponents.

With this P,

$$I_2 \times T'_2 \times I_{N/4} = P(T'_2 \times I_{N/2})P^{-1} = PCP^{-1}$$
$$I_4 \times T'_2 \times I_{N/8} = P^2CP^{-2}$$

where C is defined in (4.23). Then (4.20) becomes

$$T'_N = CE_1PCP^{-1}E_2P^2CP^{-2}E_3\ldots P^{n-1}CP^{-(n-1)} \qquad (4.26)$$

E'_i and E''_i are now defined by

$$E_i = P^{(i-1)}E'_iP^{-(i-1)} = P^iE''_iP^{-i}$$

which permits to pass the diagonal operator through the permutations. The operators E'_i and E''_i will also be diagonal, but with the values on the diagonal permuted. Using these transformations and noting that, since $P^n = 1$, $P^{-(n-1)} = P$, (4.26) becomes either

$$T'_N = CE'_1PCE'_2PCE'_3 \ldots . PCP \qquad (4.27)$$

or

$$T'_N = CPE"_1CPE"_2CPE"_3 \ldots . CP \qquad (4.28)$$

The E'_i and $E"_i$ matrices are diagonal and have the effect of multiplying selected items of the data set by various powers of w. Each of the matrices can be further factorized into diagonal operators, each of which multiplies a selected subset of the data set by a common multiplier which is a power of w. For example, for N=16, E'_1 is given by

$$E'_1 = diag(0\ 0\ 0\ 4\ 0\ 2\ 0\ 6\ 0\ 1\ 0\ 5\ 0\ 3\ 0\ 7)$$

$$= F'_1\ F'_2\ F'_4$$

where

$$F'_1 = diag(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1)$$

$$F'_2 = diag(0\ 0\ 0\ 0\ 0\ 2\ 0\ 2\ 0\ 0\ 0\ 0\ 0\ 2\ 0\ 2)$$

$$F'_4 = diag(0\ 0\ 0\ 4\ 0\ 0\ 0\ 4\ 0\ 0\ 0\ 4\ 0\ 0\ 0\ 4)$$

in exponent notation. In particular, each E'_i or $E"_i$ can be factored into the product of $F'_{(2)}$ or $F"_{(2)}$, where F' and F" multiplies exactly N/4 of the data by w^2 .

The alternative factorizations for N=16 are given by

$$F"_4 = diag(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 4\ 0\ 4\ 0\ 4\ 0\ 4)$$

$$F"_2 = diag(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 2\ 2\ 0\ 0\ 2\ 2)$$

$$F"_1 = diag(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1)$$

With these factorization of the E - operators (4.27) and (4.28) becomes

$$T'_{16} = CF'_{(4)}F'_{(2)}F'_{(1)}PCF'_{(4)}F'_{(2)}PCF'_{(4)}PCP \qquad (4.29)$$

$$T'_{16} = CPF''_{(4)}F''_{(2)}F''_{(1)}CPF''_{(4)}F''_{(2)}CPF'_{(4)}CP \qquad (4.30)$$

Since the diagonal matrices commute the subsequence of the F operators can be permuted as convenient.

The general rules for the F - matrices are as follows : We number the positions along the diagonal from 0 to (N-1), and express these numbers in binary form. Then, in $F'_{(2^-)}$, the exponent coefficients are 2^m in those positions in which the (m+1)st and the last digit are 1. All other coefficients are 0. Similarly, in general, the $F''_{(2\ )}$ matrix has the exponent coefficient 2^m in the positions for which, in the binary representation, the first and the (m+2)nd digits are 1.

The final results, given in (4.27) and (4.28), can be written as

$$T'_{N=2^m} = \prod_{m=1}^{n} (CE'_m P) \qquad (4.31)$$

$$= \prod_{m=1}^{n} (CPE''_m) \qquad (4.32)$$

where E'_m and E''_m are the diagonal matrices defined above, and $E'_n = E''_n = I$. In a parallel processor, each factor represents one "pass", pass being defined as a single sequence of parallel processing operations involving, in (4.31), first a permutation of data set; second, the multiplication of the selected elements of the data set by appropriate powers of w; and finally replacement of the data set by the sum and differences of adjacent elements.

4.4 Good Thomas Algorithm

4.4.1 Theory

The second type of FFT is the Good Thomas algorithm. The length of the data sequence is factored into relatively prime

factors. It is more difficult conceptually but like Cooley - Tukey, it relies on transformation of uni-dimensional data into multi - dimensional data. When used in conjunction with Cooley - Tukey it can lead to more efficient algorithm. The Good - Thomas algorithm rearranges the input one-dimensional data such that the transform can be done by multidimensional transform without the intervening "twiddle factors".

The derivation of the Good-Thomas FFT algorithm is based on the Chinese Remainder theorem for integers. The input index is described by its residues as follows :

$$i'' = i \pmod{n''} \tag{4.33}$$

$$i' = i \pmod{n'} \tag{4.34}$$

By Chinese Remainder Theorem there exists integers N' and N" such that the input index can be recovered as follows :

$$i = i'N'n'' + i''N''n' \tag{4.35}$$

where the integers N' and N" are the integers that satisfy

$$N'n'' + N''n' = 1 \tag{4.36}$$

The output index is described somewhat differently. Define

$$\begin{aligned} k' &= N'k \bmod n' \\ k'' &= N''k \bmod n'' \end{aligned} \tag{4.37}$$

the output index k can be recovered as follows :

$$k = n''k' + n'k'' \pmod{n} \tag{4.38}$$

Substituting the expressions for the input and output indices in

the DFT formula,

$$V_{k',k''} = \sum_{i'=0}^{n'-1} \sum_{i''=0}^{n''-1} \beta^{i'k'} \tau^{i''k''} v_{i',i''} \qquad (4.39)$$

where β and τ are actually the primary n'th and the n"th root of unity needed for the n'-point Fourier transform and the n"-point Fourier transform respectively.

The Good-Thomas algorithm can be applied as long as the factors are relatively prime. Any further factorization is not permissible. To break the problem further there is a need to use the Cooley - Tukey Algorithm along with the Good - Thomas algorithm for finding FFT for power of a prime.

The above description leads to the following algorithm :

4.4.1A Good Thomas Algorithm :

1. Read n,{x[i],i=0,1,2,...,N-1}

2. Factorize n in the form

$$n = p_1^{c} p_2^{c} \ldots p_m^{c} = n_1 \cdot n_2 \cdot \ldots n_m \qquad (4.40)$$

3. Find the values of the coefficients used in the expression of CRT.

Input index : i , i = 0,1,2, ... ,N-1

Output index : j , j = 0,1,2, ... ,N-1

$i_k = i \bmod n_k$, k=1,2,3,...,m.

$$i = \sum_{k=1}^{m} N_k \; M_k \; i_k \qquad (4.41)$$

where $M_k = N / n_k$ and $M_k \cdot N_k = 1 \bmod n_k$

Find the values of M_k and N_k by CRT algorithm described

By DFT,

$$i_k \longrightarrow j_k \qquad (4.46)$$

but

$$j_k = N_k\, j \bmod n_k \quad k=1,2,\ldots,m \qquad (4.47)$$

and

$$j = \sum_{k=1}^{m} M_k\, j_k \pmod{n} \qquad (4.48)$$

From (4.41), define

$$j' = \sum_{k=1}^{m} N_k\, M_k\, j_k$$

The element of the result sequence at the j'th position should be at the jth position.

8a. Find $j_k = j' \bmod n_k$ $\quad k=1,2,3,\ldots,N-1.$

$\qquad\qquad\qquad

frequency component and the zero time component must be treated specially. Then

$$X_0 = \sum_{i=0}^{p-1} x_i \qquad (4.48)$$

$$X_0 = x_0 + \sum_{i=1}^{p-1} x_i \cdot W_N^{ik}$$

For each i from 1 to N-1, where N=p in this case, let r(i) be the unique integer from 1 to N-1 such that in GF(p), $\pi^{r(i)} = i$. The function r(i) is a map from the set {1,2,....N-1} onto the set {1,2,...,N-1} ; it is a permutation of {1,2,...,N-1}. Then X_k can be written

$$X_{\pi^{r(k)}} = x_0 + \sum_{i=1}^{N-1} W_N^{r(i)+r(k)} \cdot x_{\pi^{r(i)}}$$

Because r(i) is a permutation, set l=r(k), set j=N-1-r(i), and use j as the index of summation to get

$$X_{\pi^{l}} = x_0 + \sum_{j=1}^{N-1} W_N^{\pi^{l-j}} \cdot x_{\pi^{n-1-j}}$$

$$X'_l = x_0 + \sum_{j=0}^{N-2} W_N^{\pi^{l-j}} \cdot x'_j$$

where $X'_l = X_{\pi^l}$ and $x'_j = x_{\pi^{n-1-j}}$ are scrambled input and output data sequences. This is now the equation of a cyclic convolution between x'_j and $W_N^{\pi^l}$. Thus, by scrambling the input and output indices, the problem of Fourier transform can be converted into a problem of convolution.

The above formulation, when the input sequence is

real/complex, results in convolution of real/complex sequence and a complex sequence. It is possible to reduce the computation effort by exploiting the symmetry of the complex sequence composed of different powers of unity. The result is summarized in the following theorem :

Theorem 4.1 : Let g(x) be a Rader polynomial, where $g_i = W_N^{\pi^i}$. For every odd prime p, the coefficients of $g(x) \pmod{x^{(p-1)/2}-1}$ are real numbers, and the coefficients of $g(x) \pmod{x^{(p-1)/2}-1}$ are imaginary numbers.

Using the above theorem the algorithm for DFT for prime length is given as follows :

4.5.1A Rader Algorithm for prime length : Given a real sequence x_i, i=0,1,2,...,N-1 where N is a prime this algorithm will compute its DFT coefficients using the Winograd algorithm for fast convolution.

1. Find the primitive element π in GF(N).
2. Set $X_0 = \sum_{i=0}^{N-1} x_i$
3. Scramble the elements of the x vector such that the element at position $j' = \pi^{N-1-j}$ is at position j for j=0,1,2,...,N-2.
4. Process the elements of the resultant vector such that the new contents are defined by the following :

$$x_i = x_i + x_{i+(N-1)/2}$$

$$x_{i+(N-1)/2} = x_i - x_{i+(N-1)/2}$$

for i=1,2,...,(N-1)/2

5. Generate a new sequence h_i , i=1,2,...,N-1 such that

$$h_i = \cos(\theta \cdot \pi^{i-1})$$

$$h_{i+(N-1)/2} = \sin(\theta \cdot \pi^{i-1}) \quad \text{for } 0 \leq i \leq (N-1)/2$$

6. Partition the vectors x and h into $x^{(1)}, x^{(2)}$ and $h^{(1)}, h^{(2)}$ respectively such that

$$x_{(i-1)}^{(1)} = x_i$$

$$x_{(i-1)}^{(2)} = x_{i+(N-1)/2}$$

$$i = 1,2,3,\ldots,(N-1)/2$$

$$h_{(i-1)}^{(1)} = h_i$$

$$h_{(i-1)}^{(2)} = h_{i+(N-1)/2}$$

$$i = 1,2,3,\ldots,(N-1)/2$$

7. Convolve $x^{(1)}$ with $h^{(1)}$ and $x^{(2)}$ with $h^{(2)}$. Let the result be in $X^{(1)}$ and $X^{(2)}$.

8. Combine the the vectors $X^{(1)}$ and $X^{(2)}$ to obtain the resultant vector X with

$$X_j = X^{(1)}_i + j \cdot X^{(2)}_{i+(N-1)/2} \ , \ 1 \leq i \leq (N-1)/2$$

$$= X^{(1)}_i - j \cdot X^{(2)}_{i+(N-1)/2} \ , \ (N-1)/2+1 \leq i \leq (N-1)$$

9. Set $X_i = x_0 + X_i$ for i = 1,2,3,...,N-1.

4.5.2 Winograd Fourier Transform Algorithm for power of an odd - prime

The idea of Rader Algorithm can be used even for sequences of length $N=p^c$, where p is a odd prime is considered. As discussed in Chapter 2 that primitive roots π modulo p^c always exists and

that these primitive roots are of order $p^{c-1}(p-1)$. Thus it is possible to convert a DFT of dimension p^c into a cyclic convolution of length $p^{c-1}(p-1)$ plus some additional terms. It can be shown that the DFT of size p^c can be partitioned into two DFTs of size p^{c-1} and one convolution of length $p^{c-1}(p-1)$. The procedure of breaking up can be used recursively to convert the DFTs of size p^{c-1} into convolutions.

One of the operation which is required for implementation is finding the primitive root modulo p^m for $m=1,2,\ldots,c$. This operation makes the algorithm computationally inefficient. The following theorem reduces the problem to just finding the primitive root modulo p.

Theorem 4.2 : If an element is a primitive root modulo p, then it is also a primitive root modulo p^c, but the converse is not true.

Using the above theorem and the discussion before that algorithms that are computationally efficient can be implemented.

CHAPTER FIVE

C O N C L U S I O N S

Various algorithms for the computation of discrete convolution and Fourier transform are implemented. The matrix based formulation of the Winograd algorithm for the computation of short convolutions are implemented for sequences of length 2, 3, 4, 5, 7 and 9. For convolution of long sequences it is necessary that the relatively prime factors of data sequence length, N, should belong to the set F = {2,3,4,5,7,9}. Also for convolution of multidimensional data, the size of each dimension should satisfy the above condition.

The mixed radix Cooley-Tukey algorithm and the Good-Thomas algorithm are quite general and can be applied for any lengths. But for using the Rader algorithm or the Winograd algorithm it is necessary that the length of resulting convolutions has its relatively prime factors belonging to F = {2,3,4,5,7,9}.

It is attempted to keep the complexity of the algorithm at the minimum without compromising the speed and the memory requirements of the programs. The trignometric functions, namely sine and cosine, are generated and stored before running any discrete Fourier transform program. This results in an appreciable increase in the running time as they are used repeatedly in the computation of discrete Fourier transform. To minimize the memory requirements of the programs the dynamic memory allocation facility of Pascal is utilized. All the intermediate variables are released back to the heap memory after their use.

All the algorithms have been written as procedures. This

allows sharing of a common set of procedures for all the algorithms. All the related procedures are kept in a single file as Pascal units. The use of units permits storing the procedures in compiled form.

5.1 Scope for future work

One of the serious restriction of the convolution algorithm implemented is that the length of the convolution should have relatively prime factors belonging to the set F={2,3,4,5,7,9}. The algorithm for convolution of sequences length of a power of an odd prime can be very useful.

All the algorithms implemented in this thesis are intended for operations on real and rational field and integer ring. The factorization of polynomials are done over rational field. Use of polynomial transforms can reduce the computational complexity of the algorithms by factorizing the modulo polynomial over polynomial extension fields. The algorithms already implemented for Winograd convolution based on the polynomial formulation can be used for further development. Also a scheme for generating the A, B and C matrices can be developed from the polynomial formulation. The resulting matrices may not be the most optimum but can serve as a starting point for further reduction of computational complexity.

APPENDIX I

L I S T O F P R O C E D U R E S

I.1 Discrete Fourier Transform

Unit Name : CompDef

Constant Declarations :

```
MaxSamp = 100;
UnitSize = 64;
NoOfTypes = 6;
NoOfFactors = 15;
NoOfPrimes  = 20;
```

Type Declarations :

```
Comp_no     = array[1..2] of real;
Comp_Vector = array[0..MaxSamp] of Comp_no;
Comp_Unit   = array[0..UnitSize] of real;
Descriptor  = array[1..NoOfTypes,1..6] of integer;
FactorArr   = array[1..NoOfFactors] of integer;
Primes      = array[1..NoOfPrimes] of integer;
```

Procedures and functions :

1. procedure TimeStart(var i:word; var j:word; var k:word;
   var Tick :real);

   Input : None

   Output : i,j,k,Tick

Records the current minute in i, current second in j and decimal second in k. The variable Tick is initialized to zero.

2. procedure TimeStop(i,j,k:word; var Tick);

Input : i,j,k

Output : Tick

Calculates the time taken in seconds between the current time and the time when i,j and k were stored using the TimeStart procedure.

Unit Name : Dft

Uses CompDef

Procedures and functions :

1. procedure Comp_Mult(a,b : Comp_no; var c:Comp_no);

Input : a,b

Output : c

Multiplies complex numbers a and b and stores the result in c.

2. procedure CmpMltIn(var a:Comp_no; b:Comp_no);

Input : a,b

Output : a

Multiplies a and b and stores the result in a.

3. procedure DftBrute(var InVector:Comp_Unit;
n,flag: integer);

Input : InVector,n,flag

Output : InVector

Computes the discrete Fourier transform of InVector of size n. If Flag = 1 the DFT is computed else if Flag = -1 the IDFT is computed.

Unit Name : PrimRoot

Uses CompDef

Procedures and functions :

1. function modsqr(var x:integer; var n integer):integer;

Computes x^2 mod n.

2. function modmult(var x:integer; var y:integer;
var n:integer):integer;

Returns x·y mod n.

3. function FastExp(var a:integer; var z:integer;
var n:integer):integer;

Returns a^z mod n.

4. function primitive(base:integer):integer;

Returns the primitive element of GF(base). It requires base to be a prime number.

Unit Name : Factor

Uses CompDef

Procedures and functions :

1. procedure Factorize(n:integer; var Factor:FactorArr;
var S_no:integer; var prime:boolean);

Input : n

Output : Factor,S_no,prime.

Prime is true if n is a prime number. The factors of n are stored in factor array and S_no indicates the number of factors.

Unit Name : Radix2

Uses CompDef

Procedures and functions

1. procedure DftRadix2(var InVector:Comp_Vector;
n,m:integer);

Input : InVector,n,m

Output : InVector

Computes the DFT of InVector of size $n = 2^m$. The result is stored back in InVector.

Unit Name : Thomas

Uses CompDef,Dft,Factor,Radix2

Procedures and functions :

1. procedure ComRadix(var Buff : Comp_Unit;
base,power:integer);

Input : Buff,base,power

Output : Buff

Computes the DFT of the vector Buff of size $base^{power}$ using the Cooley Tukey algorithm, where base is an odd prime.

2. procedure ThomasDft(var InVector:Comp_Vector;N:integer);

Input : InVector,N

Output : InVector

Computes the DFT of InVector of dimension N. The procedure utilizes the Good-Thomas Algorithm.

Unit Name : PrimeDft

Uses CompDef,Dft,PrimRoot,Factor

Procedures and functions :

1. procedure OddPrime(var InVector:Comp_Unit;
base,power : integer);

Input : base,power,InVector

Output : InVector

Computes the DFT of InVector of length N = $(base)^{power}$. The base should be a power of an odd prime.

Unit Name : WinoDef

```
Constant Declarations :
     nsize = 20;
     n1size = 20;
     n2size = 20;
     maxdim = 20;
     MaxFac = 4;
Type Declarations
     OneDim = array[0..nsize] of real;
     TwoVec = array[0..1] of onedim;
     SmlVec = array[0..MaxDim] of real;
     TwoElmt = array[1..2] of real;
     Polynom = array[0..n1size] of twoelmt;
     TwoDim  = array[0..n1size] of polynom;
     FacArr  = array[1..MaxFac] of integer;
     Fac2Arr = array[0..1] of FacArr;
```

Unit Name : ImageDef

```
Uses      WinoDef
Constant Delarations
     MaxLen1 = 40;
     MaxLen2 = 40;
Type Declarations
     ImageArr = array[0..MaxLen1,0..MaxLen2] of real;
     AllFac   = array[0..1] of Fac2Arr;
```

Unit Name : Wino2pro

Uses WinoDef

Procedures and functions :

1. procedure Wino2frd(var d: SmlVec);

   Input : d

   Output : d

Does the operation of pre - addition on the elements d vector of length 2. The resulting vector is of size M(2).

2. procedure Wino2frg(var d: SmlVec);

   Input : g

   Output : g

Does the operation of pre - addition on the elements g vector of length 2. The resulting vector is of size M(2).

3. procedure Wino2Rev(var s: SmlVec);

   Input : s

   Output : s

Does the operation of post - addition on the elements s vector of length M(2). The resulting vector is of size 2.

Algorithms for input sequences of size 3,4,5,7 and 9 are implemented. The name of the procedures are indentical to the ones listed above with 2 replaced by appropraite number.

Unit Name Winosub

Uses Winodef, Wino2pro, Wino3pro, Wino4pro, Wino5pro, Wino7pro

Procedures and functions

1. function FindMult(i : integer):integer;

Returns the number of multiplications required to compute i - point convolution using Winograd algorithm.

2. procedure WinoProc(way,NoOfPts:integer;
var original : SmlVec);

Input : way,NoOfPts,original

Output : original

Does the pre-addition or post addition operation on the elements of original vector of size NoOfPts. The operation to be done depends on the value of variable way.

Way	Operation done
1	Pre - addition corresponding to d vector.
2	Pre - addition corresponding to g vector.
3	Post addition

Unit Name : Wino2dim

Uses Winodef, Wino2pro, Wino3pro, Wino4pro, Wino5pro, Wino7pro,WinoSub

Procedures and functions :

1. procedure AgClyFr(var g: onedim; var d: onedim;
n,nn,mm : integer; var data: twodim);

Input : n,nn,mm,g,d

Output : data

Converts two one-dimensional vector of length n = nn x mm into a two - dimensional data using the Agarwal-Cooley algorithm.

2. procedure AgClyRev(var s : onedim; n,nn,mm:integer;
var data : twodim);

Input : data,n,nn,mm

Output : s

Does the inverse operation of AgClyFr ; it converts a two-dimensional data into a onedimensional data. The dimension of the data vector is nn x mm = n.

3. procedure processor(check:integer; var data : Twodim;
nn,mm :integer; var nm1 : integer; var nm2:integer);

Input : nn,mm,data

Output : nm1,nm2,data

Computes the Winograd transform along the rows and columns of data. The number of rows and columns are nn and mm respectively. The order of the resultant matrix is nm1 x nm2. If check = 1 then the 'd' transform is done, and if check=2 then 'g' transform is carried out.

3. procedure reverse(var data: Twodim; nn,mm,nm1,nm2
:integer);

Input : nn,mm,nm1,nm2,data

Output : data

Computes the Inverse Winograd transform along the rows and columns of data. The number of rows and columns are nm1 and nm2 respectively. The order of the resultant matrix is nn x mm. The operation can be viewed as the inverse of processor.

Unit Name : Winoform

Uses Winodef,Wino2pro,Wino3pro,Wino4pro,Wino5pro, Wino7pro,Winosub

Procedures and functions

1. procedure scramble(scram : byte; var Vector:onedim; nn, nofac :integer; var Factor:Fac2Arr; var weight : FacArr);

   Input : scram,nn,nofac,Factor,weight,nofac,Vector

   Output : Vector

   Scrambles the elements of the input vector Vector. The length is n = nn+1. The relatively prime factors of n are given in Factor with nofac indicating the number of relatively prime factors of n. weight indicates the weights for the mixed radix representation of input index. If scram is 0 then scrambling is done, else if it is 1 then unsrambling is done.

2. procedure transform(flag:byte; var ProVec:onedim; nofac, n, NoOfMult : integer; var Factor:Fac2Arr; var weight: FacArr);

   Input : flag,ProVec,nofac,n,NoOfMult,Factor,weight

   Output : ProVec

Computes the forward Winograd transform or the reverse Winograd transform depending on the value of flag. If flag is 1 then the 'd' transform is done ; if flag=2 then 'g' transform is done and if flag=3 then the reverse 's' tramsform is done. The meaning of other variables remain the same.

I.3 Polynomial Algebra

Unit Name PolyDef

Constant Declarations

MaxTerms = 12

Type Declarations

PolyType = array[-1..MaxTerms] of real;

IntPoly = array[-1..MaxTerms] of integer;

Unit Name : PolyInt

Uses PolyDef

Procedures and functions

1. procedure modulo(var a : intpoly ; var m :intpoly);

Input : a,m

Output : a

Computes a mod m, where a and m are integer polynomials. The result is stored back in a.

2. procedure Multiply(var a:intpoly; var b:intpoly);

Input : a,b

Output : a

Multiplies integer polynomials a and b.

3. procedure Division(var dividend : intpoly;
var divisor : intpoly;
var quotient : intpoly);

Input : dividend,divisor

Output : quotient,dividend

Divides dividend by divisor to give the quotient polynomial and the remainder is stored in dividend after execution. This procedure is suitable only for cases when the results are known to belong to ring of integer polynomials.

4. procedure PseuDiv(var ddpoly:intpoly;
var drpoly:intpoly;
var qupoly:intpoly;
var denom :integer);

Input : ddpoly,drpoly

Output : qupoly,denom

Divides integer polynomials ddpoly and drpoly. The resulting quotient polynomial is stored in qupoly and the remainder in ddpoly. Denom stores the common denominator for remainder and quotient as they can be rational in the general case.

Unit Name : Polynom

Uses PolyDef

Procedures and functions

1. procedure AddReal(var a:polytype; var b:polytype);

Input : a,b

Output : a

Addition of real polynomials a and b .

2. procedure rmod(var a:polytype; var m:intpoly);

Input : a,m

Output : a

computes a mod m.

3. procedure rmult(var a:polytype; var b:polytype);

Input : a,b

Output : a

Multiplies a and b.

4. procedure rimult(var a:polytype; var b:intpoly);

Input : a,b

Output : a

Multiplies real polynomial a with an integer polynomial b.

Unit Name : gcd

Uses PolyDef

Procedures and functions

1. function TwoGCD(x,y:integer):integer;

Returns the gcd of two integers x and y.

2. function cont(var numbers : intpoly):integer;

Returns the content of the polynomial numbers.

3. procedure PrimPoly(var numbers:intpoly; var k:integer);

Input : numbers

Output : numbers,k

Reduces numbers to primitive form and k returns the content of numbers.

4. function leader(var numbers:intpoly);

Returns the leading coefficient of the input polynomial numbers.

Unit Name : Rational

Uses PolyDef,PolyInt,gcd

Procedures and functions :

1. procedure ShowOff(n,d:integer;var a:intpoly);

   Input : n,d,a

   Output : None

Displays a rational polynomial a with the common numerator n and denominator d.

2. procedure PolyCopy(n,d:integer; var a:intpoly;
   var nn:integer; var dd:integer;
   var b : intpoly);

   Input : n,d,a

   Output : nn,dd,b

   Copies the input variables onto the output variables.

3. procedure Reduce(var i:integer; var j:integer; var
   k:integer);

   Input : i,j

   Output : i,j,k

   Sets k = gcd(i,j). Also reduces i and j such that i=i/k and j=j/k.

4. procedure AddRat(var na:integer;
   var da:integer;
   var a :intpoly;
   nb:integer;
   db:integer;
   var b :intpoly);

Input : na,da,a,nb,db,b

Output : na,da,a

Adds rational polynomials a*(na/da) and b*(nb/db).

5. procedure SubRat(var na:integer;
var da:integer;
var a :intpoly;
nb:integer;
db:integer;
var b :intpoly);

Input : na,da,a,nb,db,b

Output : na,da,a

Subtracts rational polynomials a*(na/da) and b*(nb/db).

6. procedure MultRat(var na:integer;
var da:integer;
var a :intpoly;
nb:integer;
db:integer;
var b :intpoly);

Input : na,da,a,nb,db,b

Output : na,da,a

Multiplies rational polynomials a*(na/da) and b*(nb/db).

7. procedure DivRat(var na:integer;
var da:integer;
var a :intpoly;
nb:integer;
db:integer;
var b :intpoly);
var nc:integer;
var dc:integer;
var c :intpoly);

Input : na,da,a,nb,db,b

Output : na,da,a,nc,dc,c

Divides rational polynomials a*(na/da) and b*(nb/db)

The remaider is stored in a and the quotient in c.

Unit Name : InVer

Uses PolyDef,PolyInt,gcd,rational

Procedures and functions :

1. procedure InVert(var v:intpoly; var u:intpoly;
var npp:integer;var dpp:integer;var pp:
intpoly);

Input : v,u

Output : npp,dpp,pp

Computes the inverse of v mod u.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

REFERENCES

1. Cooley, J.W. and J.W.Tukey , `An algorithm for machine calculation of complex Fourier series', Math. Comput., Vol 19 pp 297-301, [1966].

2. Thomas, L.H.,`Using a computer to solve problems in physics', Application of Digital Computers, Boston, Mass., Ginn & Co. [1963].

3. Good, I.J., `The interaction algorithm and practical Fourier analysis', J. Roy. Stat. Soc., Ser. B, Vol 20, pp 361-372, [1958]; Addendum: Vol 22, [1960], pp 372-375 (MR 21 1674, MR 23 A 4231).

4. Pease, M.C., `An adaption of the Fast Fourier Transform for parallel processing', J. ACM, Vol 15, pp 252-264, April [1968].

5. Gentleman, W.M. and G.Sande, `Fast Fourier Transforms for fun and profit', 1966 Fall Joint Computer Conf. AFIPS Proc., Vol 29, Washington D.C., Spartan 1966, pp 563-578.

6. Singleton, R.C., `An algorithm for computing the mixed radix fast fourier transform', IEEE Trans. Audio Electroaccou., Vol AU-17, pp 93-103, June, [1969].

7. Rader, C.M., `Discrete Fourier Transforms when the number of data samples is prime', Proc. IEEE, Vol 56, pp 1107-1108, June [1968].

8. Winograd., S., `Some bilinear forms whose multiplcative complexity depends on the field of constants', Math. Syst. Th., 10, pp 169-180, [1977].

9. Winograd, S., `On computing the discrete fourier transform', Math. Comp, 32, pp 175-199, [1978].

10. Nussbaumer,H.J., and P.Quandalle,'Computation of convolutions and discrete Fourier transform by polynomial transforms',IBM J. Res. Dev.,22,134-144 [1978].

11. Agarwal, R.C., and C.S.Burrus,'Number Theoretic Transforms to implement fast digital convolution',Proc IEEE, vol. 63, pp 550-560, April 1975.

12. Agarwal,R.C. and J.W.Cooley,'New algorithms for digital convolution',IEEE Trans on ASSP, vol ASSP-25, No.5, Oct. 1977.

13. Knuth, D.E., 'The Art of Computer Programming', Vol. 2, Seminumerical Algorithms, Reading Mass. , Addison Wesley 1969.

14. Blahut, R.E. , ' Fast Algorithms for Digital Signal Processing', Addison Wesley,1985.

15. Good,I.J., 'The relationship between two fast Fourier transforms',IEEE Trans C-20, 310-317.

16. Berlekamp,B.R., 'Algebraic Coding Theory', McGraw Hill, New York, 1968.

EE-1989-M-KAV-SOF